目錄

说明

- 本仓库为《编写高质量代码改善 Python 程序的 91 个建议》形成 Gitbook 的代码
- 整理成 GitBook 形式了

编写高质量代码改善 Python 程序的 91 个建议

第 1 章 引论

建议 1：理解 Pythonic 概念

对于 Pythonic 的概念，有一个更具体的指南，《The Zen of Python》（Python 之禅）。有几点非常深入人心：

- 美胜丑，显胜隐，简胜杂，杂胜乱，平胜陡，疏胜密。
- 找到简单问题的一个方法，最好是唯一的方法（正确的解决之道）
- 难以解释的实现，源自不好的注意；如果有非常棒的注意，它的实现肯定易于理解

Pythonic 的定义

遵循 Pythonic 的定义，看起来就像是伪代码。其实，所有的伪代码都可以轻易地转换为可执行的 Python 代码。

Pythonic 也许可以定义为：充分体现 Python 自身特色的代码风格。

代码风格

在语法上，代码风格要充分表现 Python 自身特色。比如说两个变量交换，利用 Python 的 packaging/unpackaging 机制，Pythonic 的代码只需要以下一行：

`a, b = b, a` 。

还有，在遍历一个容器的时候：

```
for i in a_list:
    do_sth_with(i)
```

灵活地使用迭代器是一种 Python 风格。比如说，需要安全地关闭文件描述符，可以使用以下 with 语句：

```
with open(path, 'r') as f:
    do_sth_with(f)
```

应当追求的是充分利用 Python 语法，但不应当过分地使用奇技淫巧，比如利用 Python 的 Slice 语法，可以写出如下代码：

```
a = [1, 2, 3, 4]
c = 'abcdef'
print(a[::-1])
print(c[::-1])
```

实际上，这个时候更好地体现 Pythonic 的代码时充分利用 Python 库里的 `reversed()` 函数的代码。

```
print list(reversed(a))
print list(reversed(c))
```

标准库

写 Pythonic 程序需要对标准库有充分的理解，特别是内置函数和内置数据类型。比如，对于字符串格式化，推荐这样写：

```
print '(greet) from (language).'.format(greet = "Hello world", language = 'Python')
```

`str.format()` 方法非常清晰地表明了这条语句的意图，而且模板的使用也减少了许多不必要的字符，使可读性得到了很大的提升。

Pythonic 的库或框架

编写应用程序的时候的要求会更高一些。因为编写应用程序一般需要团队合作，那么可能你编写的那一部分正好是团队的另一成员需要调用的接口，换言之，你可能正在编写库或框架。

程序员利用 Pythonic 的库或框架能更加容易、更加自然地完成任务。如果用 Python 编写的库或框架迫使程序员编写累赘的或不推荐的代码，那么可以说它并不 Pythonic。现在业内通常认为 Flask 这个框架是比较 Pythonic 的，它的一个 Hello world 级别用例如下：

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello():
    return "Hello world!"

if __name__ == '__main__':
    app.run()
```

一个 Pythonic 的框架不会对已经通过惯用法完成的东西重复发明“轮子”，而且它也遵循常用的 Python 惯例。创建 Pythonic 的框架及其困难，现在认为像 generators 之类的特性尤为 Pythonic。

另一个有关新趋势的例子是，Python 的包和模块结构日益规范化。现在的库或框架跟随了一下潮流：

- 包和模块的命名采用小写、单数形式，而且短小
- 包通常仅作为命名空间，如只包含空的 `__init__.py` 文件。

建议 2：编写 Pythonic 代码

- 要避免劣化代码，比如不合适的变量命名等
    - 避免只用大小写来区分不同的对象
    - 避免使用容易引起混淆的名称，包括：
        - 重复使用已经存在于上下文中的变量名来表示不同的类型
        - 误用了内建名称来表示其他含义的名称而使之在当前命名空间被屏蔽
        - 没有构建新的数据类型的情况下使用类似于 `element`、`list`、`dict` 等作为变量名
        - 使用 o（字母 O 小写的形式，容易与数值 0 混淆）、1（字母 L 小写的形式，容易与数字 1 混淆）等作为变量名
    - 以下两个示例，示例二比示例一好：

```
# 示例一：
def funA(list, num):
    for element in list:
        if num == element:
            return True
        else:
            pass

# 示例二：
def find_num(searchList, num):
    for listValue in searchList:
        if num == listValue:
            return True
        else:
            pass
```

    - 不要害怕过长的变量名，比如 `person_info` 比 `pi` 的可读性要强得多。
- 深入认识 Python 有助于编写 Pythonic 代码

- 全面掌握 Python 提供给我们的所有特性，包括语言特性和库特性。其中最好的学习方式应该是通读官方手册中的 `Language Reference` 和 `Library Reference` 。
- 一方面 Python 语言推荐使用大量的惯用法来完成任务；另一方面，语言的进化又趋于更好地支持惯用法。
- 深入学习业界公认的比较 Pythonic 的代码，比如 `Flask、gevent 和 requests` 等。而使用 Python 的标准库 httplib2 时，代码就非常复杂，程序员需要了解相当多的关于 `HTTP` 协议和 `Basic Auth` 的知识才能编程。
- 最后，也可以尝试利用工具达到事半功倍的效果。比如风格检查程序 PEP8。其实一开始 PEP 8 是一篇关于 Python 编码风格的指南，它提出了保持代码一致性的细节要求。它至少包括了对代码布局、注释、命名规范等方面的要求，在代码中遵循这些原则，有利于编写 Pythonic 的代码。
    - 应用程序 PEP 8 可以用来进行检测，它是使用 Python 开发的，安装的方法： `pip install -U pep8`
    - 然后即可简单地用它检测一下自己的代码： `pep8 --first optparse.py` ，如果嫌报表不够细致，可以考虑使用 `--show-source` 参数让 `PEP8` 显示每一个错误和警告对应的代码。
    - 比如，对代码的换行，不好的风格如下：

```
if foo == "blah": do_blah_thing()
do_one(); do_two(); do_three()
```

      而 Pythonic 的风格则是这样的：

```
if foo == "blah":
    do_blah_thing()
do_one()
do_two()
do_three()
```

    - `PEP8` 程序，甚至还可以给出“正确”的写法，除了针对某一个源代码文件以外，还可以直接检测一个项目的质量，并通过直观的报表给出报告。
    - `PEP8` 有优秀的插件架构，可以方便地实现特定风格的检测；它生成的报告易于处理，可以很方便地与编辑器集成。
    - `PEP8` 不是唯一的编程规范。有些公司制定的编程规范也非常有参考意义，比如 `Google Python Style Guide` 。同样， `PEP8` 也不是唯一的风格检测程序，类似的应用还有 `Pychecker、Pylint、Pyflakes` 等。其中 `Pychecker` 是 `Google`

`Python Style Guide` 推荐的工具；`Pylint` 因可以非常方便地通过编辑配置文件实现公司或团队的风格检测而受到许多人的青睐；`Pyflakes` 则因为易于集成到 vim 中，所以使用的人也非常多。

建议 3：理解 Python 与 C 语言的不同之处

Python 底层是用 C 语言实现的，但切忌用 C 语言的思维和风格来编写 Python 代码。

1. “缩进”与“{}”

   避免缩进带来的困扰的方法之一就是养成良好的习惯，统一缩进风格，不要混用 Tab 键和空格。

2. ‘与“

   C 语言中单引号代表一个字符，它实际对应于编译器所采用的字符集中的一个整数值。而双引号则表示字符串，默认以 '\0' 结尾。

3. 三元操作符 ”?:“

   `C?X:Y` 在 Python 中等价的形式为 `X if C else Y`

4. `switch...case`

   Python 中可以使用多种方式来实现 `switch...case` 比如下面这一种：

```
switch(n) {
  case 0:
    printf("0");
    break;
  case 1:
    printf("1");
    break;
  case 2:
    printf("2");
    break;
  default:
    printf("????");
    break;
}
```

```
def f(x):
    return {
      0: "0",
      1: "1",
      2: "2"
    }.get(n, "???")
```

建议 4：在代码中适当添加注释

Python 中有 3 种形式的代码注释：块注释、行注释以及文档注释（docstring）。这 3 种形式的惯用法大概有如下几种：

- 使用块或者行注释的时候仅注释那些复杂的操作、算法，还有可能别人难以理解的技巧或者不够一目了然的代码
- 注释和代码隔开一定的距离，同时在块注释之后最好多留几行空白再写代码。
- 给外部可访问的函数和方法添加文档注释。注释要清楚地描述方法的功能，并对参数、返回值以及可能发生的异常进行说明，使得外部调用它的人员仅仅看 docstring 就能正确使用。（使用 `""""""` 进行注释）
- 推荐在头文件中包含 copyright 申明、模块描述等，如有必要，可以考虑加入作者信息以及变更记录
- 需要避免的注释：
    - 代码即注释（不写注释）
    - 注释与代码重复
    - 利用注释语法快速删除代码。对于不再需要的代码，应该将其删除，而不是将其注释掉。即使担心以后还会用到，版本控制工具也可以让你轻松找回被删除的代码。
    - 代码不断更新而注释却没有更新
    - 注释比代码本身还复杂繁琐
    - 将别处的注释和代码一起拷贝过来，但上下文的变更导致注释与代码不同步
    - 将注释当作自己的娱乐空间从而留下个性特征

建议 5：通过适当添加空行使代码布局更为优雅、合理

Python 代码布局有一些基本规则可以遵循：

- 在一组代码表达完一个完整的思路之后，应该用空白行进行间隔。如每个函数之间，导入声明、变量赋值等。通俗点讲就是不要在一段代码中说明几件事。推荐在函数定义或者类定义之间空两行，在类定义与第一个方法之间，或者需要进行语义分割的地方空一行。
- 尽量保持上下文语义的易理解性
- 避免过长的代码行，每行最好不要超过 80 个字符。超过的部分可以用圆括号、方括号和花括号等进行行连接，并且保持行连接的元素垂直对齐。
- 不要为了保持水平对齐而使用多余的空格，其实使阅读者尽可能容易地理解代码所要表达的意义更重要。
- 空格的使用要能够在需要强调的时候警示读者，在疏松关系的实体间起到分隔作用，而在具有紧密关系的时候不要使用空格。具体细节如下：
    - 二元运算符（赋值、比较、布尔运算的左右两边应该有空格）
    - 逗号和分号前不要使用空格
    - 函数名和左括号之间、序列索引操作时序列名和 [] 之间不需要空格，函数的默认参

数两侧不需要空格
  - 强调前面的操作符的时候使用空格

建议 6：编写函数的几个原则

函数能够带来最大化的代码重用和最小化的代码冗余。精心设计的函数不仅可以提高程序的健壮性，还可以增强可读性、减少维护成本。

一般来说函数设计有以下基本原则可以参考：

- 原则 1：函数设计要尽量短小，嵌套层次不宜过深。最好能控制在 3 层以内。
- 原则 2：函数申明应该做到合理、简单、易于使用。参数个数不宜太多。
- 原则 3：函数参数设计应该考虑向下兼容。比如相同功能的函数不同版本的实现，唯一不同的是在更高级的版本中添加了参数导致程序中函数调用的接口发生了改变。这并不是最佳设计，更好的方法是通过加入默认参数来避免这种退化，做到向下兼容。
- 原则 4：一个函数只做一件事，尽量保证函数语句粒度的一致性。
- 原则 5：不要在函数中定义可变对象作为默认值
- 原则 6：使用异常替换返回错误
- 原则 7：保证通过单元测试

建议 7：将常量集中到一个文件

Python 的内建命名空间是支持一小部分常量的，例如 True、False、None 等，只是 Python 没有提供定义常量的直接方式而已。

在 Python 中应该如何使用常量？一般来说有以下两种方式：

- 通过命名风格来提醒使用者该变量代表的意义为常量，如常量名所有字母大写，用下划线连接各个单词

- 通过自定义的类实现常量功能。这要求符合”命名全部为大写“和”值一旦绑定便不可再修改“这两个条件。下面是一种较为常见的解决办法，它通过对常量对应的值进行修改时或者命名不符合规范时抛出异常来满足以上常量的两个条件。

```
class _const:
    class ConstError(TypeError):
        pass
    class ConstCaseError(ConstError):
        pass

    def __setattr__(self, name, value):
        if self.__dict__.has_key(name):
            raise self.ConstError, "Can't change const.{}".format(name)
        if not name.isupper():
            raise self.ConstCaseError, "const name {} is not all uppercase".format
(name)
        self.__dict__[name] = value

import sys
sys.modules[__name__] = _const()
```

如果上面的代码对应的模块名为 const，使用时候只需要 import const，便可以直接定义常量了，如以下代码：

```
import const
const.COMPANY = "IBM"
```

- 无论采用哪一种方式来实现常量，都提倡将常量集中到一个文件中，因为这样有利于维护，一旦需要修改常量的值，可以集中统一而不是逐个文件去检查。采用第二种方式实现的常量可以这么做：将存放常量的文件命名为 `constant.py` ，并在其中定义一系列的常量。

```
class _const:
    [...]

import sys
sys.modules[__name__] = _const()
import const
const.MY_CONSTANT = 1
const.MY_SECOND_CONSTANT = 2
```

当在其他模块中引用这些常量时，按照如下方式进行即可：

```
from constant import const
print(const.MY_SECOND_CONSTANT)
```

第 2 章 编程惯用法

建议 8：利用 assert 语句来发现问题

断言（assert）在很多语言中都存在，它主要为调试程序服务，能够快速方便地检查程序的异常或者发现不恰当的输入等，可防止意想不到的情况出现。

其基本语法如下：

`assert expression1, [","expression2]`，使用的例子如下：

```
x, y = 1, 2
assert x == y, "not equal"
```

在执行过程中它实际相当于如下代码：

```
x, y = 1, 2
if __debug__ and not x == y:
    raise AssertionError("not equals")
```

对 Python 中使用断言需要说明如下：

- `__debug__` 的值默认设置为 True，且是只读的，在 Python 2.7 中还无法修改该值。
- 断言是有代价的，它会对性能产生一定的影响，对于编译型的语言这也许并不那么重要，因为断言只在调试模式下启用。但 Python 并没有严格定义调试和发布模式之间的区别，通常禁用断言的方法是在运行脚本的时候加上 `-O` 标志，这种方式带来的影响是它并不优化字节码，而是忽略与断言相关的语句。

断言实际是被设计用来捕获用户所定义的约束的，而不是用来捕获程序本身错误的，因此使用断言需要注意以下几点：

- 不要滥用。若由于断言引发了异常，通常代表程序中存在 bug。因此断言应该使用在正常逻辑不可到达的地方或正常情况下总是为真的场合。

- 如果 Python 本身的异常能够处理就不要再使用断言。如对于类似于数组越界、类型不匹配、除数为 0 之类的错误，不建议使用断言来进行处理。

  ```
  # 下面的断言是多余的， 如果类型不匹配本身就会抛出异常
  def stradd(x, y):
      assert isinstance(x, basestring)
      assert isinstance(y, basestring)
      return x + y
  ```

- 不要使用断言来检查用户的输入。如对于一个数字类型，如果根据用户的设计该值的范围应该是 2~10，较好的做法是使用条件判断，并在不符合条件的时候输出错误提示信息。
- 在函数调用后，当需要确认返回值是否合理时可以使用断言
- 当条件是业务逻辑继续下去的先决条件时可以使用断言，比如 `list1` 和其副本 `list2`，如果由于某些不可控的因素，如使用了浅拷贝而 `list1` 中含有可变对象等，就可以使用断言来判断这两者的关系。

建议 9：数据交换值的时候不推荐使用中间变量

交换两个变量的值，熟悉的代码如下：

```
temp = x
x = y
y = temp
```

实际上，在 Python 中有 Pythonic 的实现方式，代码如下：

`x, y = y, x`

上面的实现方式不需要借助任何中间变量并且能够获取更好的性能（可以用 `timeit` 测试）。

之所以更优，是因为 Python 表达式计算的顺序。一般情况下 Python 表达式的计算顺序是从左到右，但遇到表达式赋值的时候表达式右边的操作数先于左边的操作数计算，其在内存中执行的顺序如下：

- 先计算右边的表达式 y, x，因此先在内存中创建元组（y, x），其标示符合值分别为 y、x 及其对应的值，其中 y 和 x 是在初始化时已经存在于内存中的对象。
- 计算表达式左边的值并进行赋值，元组被依次分配给左边的标示符，通过解压缩（unpacking），元组第一标识符（为 y）分配给左边第一个元素（此时为 x），元组第二个标识符（为 x）分配给第二个元素（为 y），从而达到实现 x、y 值交换的目的。

Python 的字节码是一种类似汇编指令的中间语言，但是一个字节码指令并不是对应一个机器指令。通过 dis 模块可以进行分析：

```
import dis
def swap1():
    x, y = 2, 3
    x, y = y, x

def swap2():
    x, y = 2, 3
    temp = x
    x = y
    y = temp

dis.dis(swap1)
dis.dis(swap2)
```

通过字节码可以看出，swpa1 对应的字节码中有 2 个 `LOAD_FAST` 指令、2 个 `STORE_FAST` 指令和 1 个 `ROT_TWO` 指令，而 swap2 函数对应的共生成了 3 个 `LOAD_FAST` 指令和 3 个 `STORE_FAST` 指令。而指令 `ROT_TWO` 的主要作用是交换两个栈的最顶层元素，它比执行一个 `LOAD_FAST + STORE_FAST` 指令更快。

建议 10：充分利用 Lazy evaluation 的特性

Lazy evaluation 常被译为“延迟计算”或“惰性计算”，指的是仅仅在真正需要执行的时候才计算表达式的值。充分利用 Lazy evaluation 的特性带来的好处主要体现在以下两个方面：

- 避免不必要的计算，带来性能上的提升。对于 Python 中的条件表达式 `if x and y`，在 x 为 false 的情况下 y 表达式的值将不再计算。而对于 `if x or y`，当 x 的值为 true 的时候将直接返回，不再计算 y 的值。因此编程中应该充分利用该特性。例如：

```
from time import time
t = time()
abbreviations = ["cf.", "e.g.", "ex.", "etc.", "flg."]
for i in xrange(100000):
    for w in ("Mr.", "Hat", "is", "chasing", "."):
        if w in abbreviations and w[-1]=='.': # 这句性能较差
        # if w[-1] == '.' and w in abbreviations: # 性能好
            pass
print time() - t
```

  如果使用注释的那一条 if 语句，运行的时间大约会节省 10%。总结来说，对于 or 条件表达式应该将值为真可能性较高的变量写在 or 的前面，而 and 则应该推后。

- 节省空间，使得无限循环的数据结构成为可能。Python 中最典型的使用延迟计算的例子就是生成器表达式了。比如斐波那契：

```
def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
from itertools import islice
print list(islice(fib(), 5))
```

建议 11：理解枚举替代实现的缺陷

枚举最经典的例子是季节和星期，它能够以更接近自然语言的方式来表达数据，使得程序的可读性和可维护性大大提高。但是枚举类型在 Python 3.4 以前却并不提供。人们充分利用 Python 的动态性这个特征，想出了枚举的各种替代实现方式：

- 使用类属性

```
class Seasons:
    Spring, Summer, Autumn, Winter = 0, 1, 2, 3
# 或者可以简化为：
class Seasons:
    Spring, Summer, Autumn, Winter = range(4)
```

- 借助函数

```
def enum(*posarg, **keysarg):
    return type("Enum", (object,), dict(zip(posarg, xrange(len(posarg))), **keysar
g))

Seasons = enum("Spring", "Summer", "Autumn", Winter=1)
```

- 使用 `collections.namedtuple`

```
Seasons = namedtuple("Seasons", "Spring Summer Autumn Winter")._make(range(4))
```

显然，这些替代实现有其不合理的地方：

- 允许枚举值重复，比如在 `collections.namedtuple` 中，使得枚举值 Spring 和 Autumn 相等，却不会提示任何错误：`Seasons._replace(Spring = 2)`
- 支持无意义的操作，比如 `Seasons.Summer + Seasons.Autumn == Season.Winter`

实际上 Python 2.7 以后的版本还有另外一种替代选择——使用第三方模块 `flufl.enum`，它包含两种枚举类：一种是 `Enum`，只要保证枚举值唯一即可，对值的类型没限制；还有一种是 `IntEnum`，其枚举值是 int 型。

```
from flufl.enum import Enum
class Seasons(Enum): # 继承自 Enum 定义枚举
    Spring = "Spring"
    Summer = 2
    Autumn = 3
    Winter = 4
Seasons = Enum("Seasons", "Spring Sumter Autumn Winter")
```

`flufl.enum` 提供了 `__members__` 属性，可以对枚举名称进行迭代。

```
for member in Seasons.__members__:
    print member
```

可以直接使用 value 属性获取枚举元素的值，比如：

`print Seasons.Summer.value`

`flufl.enum` 不支持枚举元素的比较。比如不支持 `Seasons.Summer < Seasons.Autumn`

Python3.4 中根据 PEP435 加入了枚举 Enum，其实现主要参考 `flufl.enum`，但两者之间还是存在一些差别，如 `flufl.enum` 允许枚举继承，而 `Enum` 仅在父类没有任何枚举成员的时候才允许继承等。如果要在 Python3.4 之前的版本中使用枚举 Enum，可以安装 Enum 的向后兼容包 enum34。

建议 12：不推荐使用 type 来进行类型检查

作为动态性的强类型脚本语言，Python 中的变量在定义的时候并不会指明具体类型，Python 解释器会在运行时自动进行类型检查并根据需要进行隐式类型转换。按照 Python 的理念，为了充分利用其动态性的特征是不推荐进行类型检查的。解释器能够根据变量类型的不同而调用合适的内部方法进行处理，而当 a、b 类型不同而两者之间又不能进行隐式类型转换时便抛出 TypeError 异常。

不刻意进行类型检查，而是在出错的情况下通过抛出异常来进行处理，这是较为常见的方式。但实际应用中为了提高程序的健壮性，仍然会面临需要进行类型检查的情景。

内建函数 `type(object)` 用于返回当前对象的类型，因此可以通过与 Python 自带模块 types 中所定义的名称进行比较，根据其返回值确定变量类型是否符合要求。

所有基本类型对应的名称都可以在 `types` 模块中找到，然而使用 `type()` 函数并不适合用来进行变量类型检查。

- 基于内建类型扩展的用户自定义类型，type 函数并不能准确返回结果
- 在古典类中，所有类的实例的 type 值都相等

对于内建的基本类型来说，使用 `type()` 进行类型检查问题不大，但在某些特殊场合 `type()` 方法并不可靠。解决方法是，如果类型有对应的工厂函数，可以使用工厂函数对类型做相应转换，否则可以使用 `isinstance()` 函数来检测。

```
isinstance(object, classinfo)
# 其中，classinfo 可以为直接或间接类名、基本类型名称或者由它们组成的元组，该函数在 classinfo 参数错误的情况下会抛出 TypeError 异常。
# isinstance 基本用法举例如下：
>>> isinstance(2, float)
False
>>> isinstance("a", (str, unicode))
True
>>> isinstance((2, 3), (str, list, tuple)) # 支持多种类型列表
True
```

建议 13：尽量转换为浮点类型后再做除法

- 【PS：这应该是 Python2 中存在的问题】

Python 在最初的设计过程中借鉴了 C 语言的一些规则，比如选择 C 的 long 类型作为 Python 的整数类型，double 作为浮点类型等。同时标准的算术运算，包括除法，返回值总是和操作数类型相同。作为静态类型语言，C 语言中这一规则问题不大，因为变量都会预先申明类型，当类型不符的时候，编译器也会尽可能进行强制类型转换，否则编译会报错。但 Python 作为一门高级动态语言并没有类型申明这一说。

Python 中除了除法运算之外，整数和浮点数的其他操作行为还是一致的，因此这容易让人产生一种误解，数值的计算与具体操作数的类型无关，但事实上对于整数除法这是编程过程中潜在的一个危险。推荐的做法之一是当涉及除法运算的时候尽量先将操作数转换为浮点类型再做运算。

在 Python3 中这个问题已经不存在了。Python3 之前的版本可以通过 `from __future__ import division` 机制使整数除法不再截断，这样即使不进行浮点类型转换，输出结果也是正确的。

还需要说明一点，浮点数可能是不准确的，比如：

```
i = 1
while i != 1.5:
    i = i + 0.1
    print i
```

这段代码会导致无限循环，在内存中根据浮点数位数规定，多余部分直接截断。对于浮点数的处理，要记住其运算结果可能并不是完全准确的。如果计算对精度要求较高，可以使用 Decimal 来进行处理或者将浮点数尽量扩大为整数，计算完毕之后再转换回去。而对于在 while 中使用 `i != 1.5` 这种条件表达式更是要避免的，浮点数的比较同样最好能够指明精度。

建议 14：警惕 `eval()` 的安全漏洞

Python 中 `eval()` 函数将字符串 str 当成有效的表达式来求值并返回计算结果。其函数声明如下：`eval(expression[, globals[, locals]])`。

其中，参数 globals 为字典形式，locals 为任何映射对象，它们分别表示全局和局部命名空间。如果传入 globals 参数的字典中缺少 `__builtins__` 的时候，当前的全局命名空间将作为 globals 参数输入并且在表达式计算之前被解析。locals 参数默认与 globals 相同，如果两者都省略的话，表达式将在 `eval()` 调用的环境中执行。

eval 存在安全漏洞，一个简单的例子：

```
import sys
from math import *
def ExpCalcBot(string):
    try:
        print "Your answer is", eval(user_func) # 计算输入的值
    except NameError:
        print "The expression you enter is not valid"
print 'Hi, I am ExpCalcBot. please input your expression or enter e to end'
inputstr = ''
while True:
    print 'Please enter a number or operation. Enter c to complete. :'
    inputstr = raw_input()
    if inputstr == str('e'): # 遇到输入为 e 的时候退出
        sys.exit()
    elif repr(inputstr) != repr(''):
        ExpCalcBot(inputstr)
        inputstr = ''
```

由于网络环境下运行它的用户并非都是可信任的，比如输入 `__import__("os").system("dir")`，它会显示当前目录下的所有文件列表；输入 `__import__("os").system("del * /Q")`，会导致当前目录下的所有文件都被删除了，而这一切没有任何提示。

如果在 globals 参数中禁止全局命名空间的访问：

```
def ExpCalcBot(string):
    try:
        math_fun_list = ["acos", "asin", "atan", "cos", "e", "log", "log10", "pi", "po
w", "sin", "sqrt", "tan"]
        math_fun_dict = dict([(k, globals().get(k)) for k in math_fun_list]) # 形成可以
访问的函数的字典
        print "Your name is", eval(string, {"__builtins__": None}, math_fun_dict)
    except NameError:
        print "The expression you enter is not valid"
```

再次进行恶意输入：`[c for c in ().__class__.__bases__[0].__subclasses__() if c.__name__ == "Quitter"][0](0)()`，`# ().__class__.__bases__[0].__subclasses__()` 用来显示 object 类的所有子类。类 Quitter 与 "quit" 功能绑定，因此上面的输入会导致程序退出。

注：可以在 Python 的安装目录下的 `Lib\site.py` 中找到其类的定义。也可以在解释器中输入查看输出结果。

对于有经验的侵入者来说，他可能会有一系列强大的手段，使得 eval 可以解释和调用这些方法，从而带来更大的破坏。此外，`eval()` 函数也给程序的调试带来一定困难，要查看 `eval()` 里面表达式具体的执行过程很难。因此在实际应用过程中如果使用对象不是信任源，应该避免使用 eval，在需要使用 eval 的地方可用安全性更好的 `ast.literal_eval` 替代。

建议 15：使用 `enumerate()` 获取序列迭代的索引和值

有 N 种实现方法，举例如下：

```
# 方法一：在每次循环中对索引变量进行自增
li = ['a', 'b', 'c', 'd']
index = 0
for i in li:
    print "index:", index, "element:", i
    index += 1

# 方法二：使用 range() 和 len() 方法结合
li = ['a', 'b', 'c', 'd', 'e']
for i in range(len(li)):
    print "index:", i, "element:", li[i]

# 方法三：使用 while 循环，用 len() 获取循环次数
li = ['a', 'b', 'c', 'd', 'e']
index = 0
while index < len(li):
    print "index:", index, "element:", li[index]
    index += 1

# 方法四：使用 zip() 方法
li = ['a', 'b', 'c', 'd', 'e']
for i, e in zip(range(len(li)), li):
    print "index:", i, "element:", e

# 方法五：使用 enumerate() 获取序列迭代的索引和值
li = ['a', 'b', 'c', 'd', 'e']
for i, e in enumerate(li):
    print "index:", i, "element:", e
```

推荐使用函数 `enumerate()` ，主要是为了解决在循环中获取索引以及对应值的问题。它具有一定的惰性（lazy），每次仅在需要的时候才会产生一个（index, item）对。其函数签名如下： `enumerate(sequence, start=0)`

其中，sequence 可以为序列，如 list、set 等，也可以为一个 iterator 或者任何可以迭代的对象，默认的 start 为 0，函数返回本质上为一个迭代器，可以使用 next() 方法获取下一个迭代元素。

`enumerate()` 函数的内部实现非常简单， `enumerate(sequence, start=0)` 实际相当于如下代码：

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

因此利用这个特性用户还可以实现自己的 `enumerate()` 函数。比如， `myenumerate()` 以反序的方式获取序列的索引和值。

```
def my_enumerate(sequence):
    n = -1
    for elem in reversed(sequence):
        yield len(sequence) + n, elem
        n -= 1
```

需要提醒的是，对于字典的迭代循环， `enumerate()` 函数并不适合，虽然在使用上并不会提示错误，但输出的结果与期望的大相径庭，这是因为字典默认被转换成了序列进行处理。要获取迭代过程中字典的 key 和 value，应该使用 `iteritems` 方法。

建议 16：分清 == 与 is 的适用场景

可以通过 `id()` 函数来看看变量在内存中具体的存储空间。

is 表示的是对象标示符（object identity），而 == 表示的意思是相等（equal）。is 的作用是用来检查对象的标示符是否一致的，也就是比较两个对象在内存中是否拥有同一块内存空间，它并不适合用来判断两个字符串是否相等。 `x is y` 仅当 x 和 y 是同一个对象的时候才返回 True， `x is b` 基本相当于 `id(x) == id(y)` 。而 == 才是用来检验两个对象的值是否相等的，它实际调用内部 `__eq__()` 方法，因此 `a == b` 相当于 `a.__eq__(b)` ，所以 == 操作符也是可以被重载的，而 is 不能被重载。一般情况下，如果 `x is y` 为 True 的话 `x == y` 的值也为 True（特殊情况除外，如 NaN， `a = floag('NaN')` ， `a is a` 为 True， `a == a` 为 false）。

Python 中存在 `string interning` （字符串驻留）机制，对于较小的字符串，为了提高系统性能会保留其值的一个副本，当创建新的字符串的时候直接指向该副本即可。

建议 17：考虑兼容性，尽可能使用 Unicode

Python 内建的字符串有两种类型：`str` 和 `Unicode` ，它们拥有共同的祖先 `basestring` 。其中， `Unicode` 是 Python2.0 中引入的一种新的数据类型，所有的 `Unicode` 字符串都是 `Unicode` 类型的实例。

```
# 创建一个 Unicode 字符
str_unicode = u"unicode" # 前面加 u 表示 Unicode
```

Unicode 编码系统可以分为编码方式和实现方式两个层次。在编码方式上，分为 `UCS-2` 和 `UCS-4` 两种方式， `UCS-2` 用两个字节编码， `UCS-4` 用 4 个字节编码。目前实际应用的统一码对应于 `UCS-2` ，使用 16 位的编码空间。一个字符的 `Unicode` 编码是确定的，但是在实际传输过程中，由于系统平台的不同以及出于节省空间的目的，实现方式有所差异。Unicode 的实现方式称为 Unicode 转换格式（Unicode Transformation Format），简称为 UTF，包括 `UTF-7` 、 `UTF-16` 、 `UTF-32` 、 `UTF-8` 等，其中较为常见的为 `UTF-8` 。 `UTF-8` 的特点是对不同范围的字符使用不同长度的编码，其中 `0x00 ~ 0x7F` 的字符的 `UTF-8` 编码与 ASCII 编码完全相同。 `UTF-8` 编码的最大长度是 4 个字节，从 Unicode 到 `UTF-8` 的编码方式如下所示：

Unicode 编码（十六进制）	UTF-8 字节流（二进制）
000000 ~ 00007F	0xxxxxxx
000080 ~ 0007FF	110xxxxx 10xxxxxx
000800 ~ 00FFFF	1110xxxx 10xxxxxx 10xxxxxx
010000 ~ 10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Python 中处理中文字符经常会遇到的几个问题：

- 读出文件的内容显示为乱码
- 当 Python 源文件中包含中文字符的时候抛出 SyntaxError 异常
- 普通字符和 Unicode 进行字符串连接的时候抛出 UnicodeDecodeError 异常

对字符串进行解码和编码，其中 `decode()` 方法将其他编码对应的字符串解码成 Unicode，而 `encode()` 方法将 Unicode 编码转换为另一种编码，Unicode 作为转换过程中的中间编码。 `decode()` 和 `encode()` 方法的函数形式如下：

```
# str.decode([编码参数 [, 错误处理]])
# str.encode([编码参数 [, 错误处理]])
```

常见的编码参数：

编码参数	描述
ascii	7 位 ASCII 码
laten-1 or iso-8859-1	ISO 8859-1，Latin-1
utf-8	8 位可变长度编码
utf-16	16 位可变长度编码
utf-16-le	UTF-16，little-endian 编码
utf-16-be	UTF-16，big-endian 编码
unicode-escapse	与 unicode 文字 u'string' 相同
raw-unicode-escape	与原始 Unicode 文字 ur'string' 相同

错误处理参数有以下 3 种常用方式：

- strict：默认处理方式，编码错误抛出 UnicodeError 异常
- ignore：忽略不可转换字符
- replace：将不可转换字符用 ? 代替

有些软件在保存 `UTF-8` 编码的时候，会在文件最开始的地方插入不可见的字符 BOM（ `0xEF 0xBB 0xBF` ，即 BOM），这些不可见字符无法被正确的解析，而利用 codecs 模块可以方便地处理这种问题。

```
import codecs
content = open("test.txt", "r").read()
filehandle.close()
if content[:3] == codecs.BOM_UTF8: # 如果存在 BOM 字符则去掉
    content = content[3:]
print content.decode("utf-8")
```

关于 BOM：

Unicode 存储有字节序的问题， `UTF-16` 以两个字节为编码单元，在字符的传送过程中，为了标明字节的顺序，Unicode 规范中推荐使用 BOM（Byte Order Mark）：即在 UCS 编码中用一个叫做 `ZERO WIDTH NO-BREAK SPACE` 的字符，它的编码是 `FEFF` （该编码在 UCS 中不存在对应的字符），UCS 规范建议在传输字节流前，先传输字符 `ZERO WIDTH NO-BREAK SPACE` 。这样如果接收者收到 FEFF，就标明这个字节流是 `Big-Endian` 的；如果收到 FFFE，就表明这个字节流是 `Little-Endian` 的。 `UTF-8` 使用字节来编码，一般不需要 BOM 来表明字节顺序，但可以用 BOM 来表明编码方式。字符 `ZERO WIDTH NO-BREAK SPACE` 的 `UTF-8` 编码是 `EF BB BF` 。所以如果接收者收到以 `EF BB BF` 开头的字节流，就知道这是 `UTF-8` 编码了。

Python 中的默认编码，可以通过 `sys.getdefaultencoding()` 来验证）。当调用 print 方法输出的时候会隐式地进行从 ASCII 到系统默认编码（Windows 上为 CP936）的转换。要避免这种错误需要在源文件中进行编码声明，声明可用正则表达式：`coding=[:=]\s*([-\w.]+)` 表示。一般来说进行源文件编码声明有以下三种方式：

- `# coding=<encoding name>`，比如 `#coding=utf-8`
- 第二种

```
#!/usr/bin/python
# -*- coding: <encoding name> -*-
```

- 第三种

```
#!/usr/bin/python
# vim: set fileencoding=<encoding name> :
```

Python2.6 之后可以通过 `import unicode_literals` 自动将定义的普通字符识别为 Unicode 字符串，这样字符串的行为将和 Python3 中保持一致。

建议 18：构建合理的包层次来管理 module

本质上每一个 Python 文件都是一个模块，使用模块可以增强代码的可维护性和可重用性。但在大的项目中将所有的 Python 文件放在一个目录下并不是一个值得推荐的做法，需要合理地组织项目的层次来管理模块，这就是包（Package）发挥功效的地方了。

简单说包即是目录，但与普通目录不同，它除了包含常规的 Python 文件（也就是模块）以外，还包含一个 `__init__.py` 文件，同时它允许嵌套。

包中的模块可以通过"."访问符进行访问，即"包名.模块名"。有以下几种导入方法：

- 直接导入一个包：`import Package`
- 导入子模块或子包，包嵌套的情况下可以进行嵌套导入，具体如下：

```
from Package import Module1
import Package.Module1
from Package import Subpackage
import Package.Subpackage
from Package.Subpackage import Module1
import Package.Subpackage.Module1
```

`__init__.py` 最明显的作用就是使包和普通目录区分；其次可以在该文件中申明模块级别的 `import` 语句从而使其变成包级别可见。如果 `__init__.py` 文件为空，当意图使用 `from Package import *` 将包 Package 中所有的模块导入当前名字空间时并不能使得导入的模块生效，这是因为不同平台间的文件的命名规则不同，Python 解释器并不能正确判定模块在对应的平台该如何导入，因此它仅仅执行 `__init__.py` 文件，如果要控制模块的导入，则需要对 `__init__.py` 文件做修改。

`__init__.py` 文件还有一个作用就是通过在该文件中定义 `__all__` 变量，控制需要导入的子包或者模块。之后再运行 `from ... import *`，可以看到 `__all__` 变量中定义的模块和包被导入当前名字空间。

包的使用能够带来以下便利：

- 合理组织代码，便于维护和使用
- 能够有效地避免名称空间冲突

如果模块包含的属性和方法存在同名冲突，使用 `import module` 可以有效地避免名称冲突。在嵌套的包结构中，每一个模块都以其所在的完整路径作为其前缀，因此，即使名称一样，但由于模块所对应的其前缀不同，因此不会产生冲突。

第 3 章 基础语法

建议 19：有节制地使用 `from ... import` 语句

Python 提供了 3 种方式来引入外部模块：`import` 语句、`from ... import ...` 及 `__import__` 函数。其中较为常见的为前面两种，而 `__import__` 函数与 `import` 语句类似，不同点在于前者显式地将模块的名称作为字符串传递并赋值给命名空间的变量。

在使用 `import` 的时候注意以下几点：

- 一般情况下尽量优先使用 `import a` 形式
- 有节制地使用 `from a import B` 形式
- 尽量避免使用 `from a import *`，因为这会污染命名空间

Python 在初始化运行环境的时候会预先加载一批内建模块到内存中，这些模块相关的信息被存放在 `sys.modules` 中。读者导入 sys 模块后在 Python 解释器中输入 `sys.modules.items()` 便可显示所有预加载模块的相关信息。当加载一个模块的时候，解释器实际上要完成以下动作：

- 在 `sys.modules` 中进行搜索看看该模块是否已经存在，如果存在，则将其导入到当前局部命名空间，加载结束
- 如果在 `sys.modules` 中找不到对应模块的名称，则为需要导入的模块创建一个字典对象，并将该对象信息插入 `sys.modules` 中
- 加载前确认是否需要对模块对应的文件进行编译，如果需要则先进行编译
- 执行动态加载，在当前模块的命名空间中执行编译后的字节码，并将其中所有的对象放入模块对应的字典中

对于用户定义的模块，import 机制会创建一个新的 module 将其加入当前的局部命名空间中，与此同时，`sys.modules` 也加入了该模块的相关信息。但从 id 输出的结果可以看出，本质上是引用同一个对象。同时会发现所在的目录下多了一个 `.pyc` 的文件，该文件为解释器生成的模块对应的字节码，从 `import` 之后的输出可以看出模块同时被执行。

需要注意的是，直接使用 `import` 和使用 `from a import B` 形式这两者之间存在一定的差异，后者直接将 B 暴露于当前局部空间，而将 a 加载到 `sys.modules` 集合。

对于 `from a import ...` 无节制的使用会带来什么问题：

- 命名空间的冲突
    - 一般来说在非常明确不会造成命名冲突的前提下，以下几种情况下可以考虑使用 `from ... import` 语句：
        - 当只需要导入部分属性或方法时
        - 模块中的这些属性和方法访问频率较高导致使用 "模块名.名称" 的形式进行访问

过于繁琐时
    - 模块的文档明确说明需要使用 `from ... import` 形式，导入的是一个包下面的子模块，且使用 `from ... import` 形式能够更为简单和便利时。
- 循环嵌套导入的问题
    - 解决循环嵌套导入问题的一个方法是直接使用 import 语句

建议 20：优先使用 `absolute import` 来导入模块

解释器默认先从当前目录下搜索对应的模块，当搜到时便停止搜索进行动态加载。

在 Python2.4 以前默认为隐式的 `relative import`，局部范围的模块将覆盖同名的全局范围的模块。如果要使用标准库中同名的模块，需要深入考察 `sys.modules`。Python2.5 中后虽然默认的仍然是 `relative import`，但它为 `absolute import` 提供了一种新的机制，在模块中使用 `from __future__ import absolute_import` 语句进行说明后再进行导入。同时它还通过点号提供了一种显式进行 `relative import` 的方法，"." 表示当前目录，".." 表示当前目录的上一层目录。

使用显式 `relative import` 之后再运行程序可能遇到这种错误 "ValueError: Attempted relative import in non-package"。这个问题产生的原因在于 `relative import` 使用模块的 `__name__` 属性来决定当前模块在包的顶层位置，而不管模块在文件系统中的实际位置。而在 `relative import` 的情形下，`__name__` 会随着文件加载方式的不同而发生改变。`python -m` 中 `-m` 的作用是使得一个模块像脚本一样运行。而无论以何种方式加载，当在包的内部运行脚本的时候，包相关的结构信息都会丢失，默认当前脚本所在的位置为模块在包中的顶层位置，因此便会抛出异常。

解决办法：

- 在包的顶层目录中加入参数 `-m` 运行该脚本
- 利用 Python2.6 在模块中引入的 `__package__` 属性，设置 `__package__` 之后，解释器会根据 `__package__` 和 `__name__` 的值来确定包的层次结构。

一个参考示例，以下不会出现在包结构内运行模块对应的脚本时出错的情况：

```
if __name__ == "__main__" and __package__ is None:
    import sys
    import os.path
    sys.path[0] = os.path.abspath("./../../")
    print sys.path[0]
    import app.sub1
    __package__ = str("app.sub1")
    from . import string
```

相比于 `absolute import`，`relative import` 在实际应用中反馈的问题较多，因此推荐优先使用 `absolute import`。`absolute import` 可读性和出现问题后的可跟踪性都更好。当项目的包层次结构较为复杂的时候，显式 `relative import` 也是可以接受的，由于命名冲突的原因以及语义模糊等原因，不推荐使用隐式地 `relative import`，并且它在 Python3 中已经被移除。

建议 21：`i += 1` 不等于 `++i`

Python 解释器会将 `++i` 操作解释为 `+(+i)`，其中 `+` 表示正数符号。对于 `--i` 操作也是类似。

因此需要明白 `++i` 在 Python 中语法上是合法的，但并不是我们理解的通常意义上的自增操作。

建议 22：使用 with 自动关闭资源

对文件操作完成后应该立即关闭它们，因为打开的文件不仅会占用系统资源，而且可能影响其他程序或者进程的操作，甚至会导致用户期望与实际操作结果不一致。

Python 提供了 `with` 语句，语法为：

```
with 表达式 [as 目标]:
    代码块
```

`with` 语句支持嵌套，支持多个 `with` 子句，它们两者可以相互转换。`with expr1 as e1, expr2 as e2` 与下面的嵌套形式等价：

```
with expr1 as e1:
    with expr2 as e2:
```

`with` 语句可以在代码块执行完毕后还原进入该代码块时的现场。包含有 `with` 语句的代码块的执行过程如下：

- 计算表达式的值，返回一个上下文管理器对象
- 加载上下文管理器对象的 `__exit__()` 方法以备后用
- 调用上下文管理器对象的 `__enter__()` 方法
- 如果 `with` 语句中设置了目标对象，则将 `__enter__()` 方法的返回值赋值给目标对象
- 执行 `with` 中的代码块
- 如果步骤 5 中代码正常结束，调用上下文管理其对象的 `__exit__()` 方法，其返回值直接忽略
- 如果步骤 5 中代码执行过程中发生异常，调用上下文管理器对象的 `__exit__()` 方法，并将异常类型、值及 `traceback` 信息作为参数传递给 `__exit__()` 方法。如果

`__exit__()` 返回值为 false，则异常会被重新抛出；如果其返回值为 true，异常被挂起，程序继续执行。

在文件处理时使用 with 的好处在于无论程序以何种方式跳出 with 块，总能保证文件被正确关闭。实际上，它不仅仅针对文件处理，针对其他情景同样可以实现运行时环境的清理与还原，如多线程编程中的锁对象的管理。`with` 得益于一个称为上下文管理器（context manager）的东西，用来创建一个运行时的环境。上下文管理器是这样一个对象：它定义程序运行时需要建立的上下文，处理程序的进入和退出，实现了上下文管理协议，即在对象中定义 `__enter__()` 和 `__exit__()` 方法。其中：

- `__enter__()`：进入运行时的上下文，返回运行时上下文相关的对象，`with` 语句中会将这个返回值绑定到目标对象。
- `__exit__(exception_type, exception_value, traceback)`：退出运行时的上下文，定义在块执行（或终止）之后上下文管理器应该做什么。它可以处理异常、清理现场或者处理 `with` 块中语句执行完成之后需要处理的动作。

实际上任何实现了上下文协议的对象都可以称为一个上下文管理器，文件也是实现了这个协议的上下文管理器，它们都能够与 `with` 语句兼容。

用户也可以定义自己的上下文管理器来控制程序的运行，只要实现上下文协议便能够和 `with` 语句一起使用：

```
class MyContextManager(object):
    def __enter__(self): # 实现 __enter__ 方法
        print "entering..."
    def __exit__(self, exception_type, exception_value, traceback):
        print "leaving"
        if exception_type is None:
            print "no exceptions!"
            return False
        elif exception_type is ValueError:
            print "value error!!!"
            return True
        else:
            print "other error"
            return True

with MyContextManager():
    print "Testing..."
    raise(ValueError) # 注释这一句会得到不同的效果
```

因为上下文管理器主要作用于资源共享，因此在实际应用中 `__enter__()` 和 `__exit__()` 方法基本用语资源分配以及释放相关的工作，如打开/关闭文件、异常处理、断开流的连接、锁分配等。为了更好地辅助上下文管理，Python 还提供了 `contextlib` 模块，该模块是通过

`Generator` 实现的，`contextlib` 中的 `contextmanager` 作为装饰器来提供一种针对函数级别的上下文管理机制，可以直接作用于函数/对象而不用去关心 `__enter__()` 和 `__exit__()` 方法的具体实现。

建议 23：使用 else 子句简化循环（异常处理）

在循环中，`else` 子句提供了隐含的对循环是否由 break 语句引发循环结束的判断。例子：

```
# 以下两段代码等价
## 我们借助了一个标志量 found 来判断循环结束是不是由 break 语句引起的。
def print_prime(n):
    for i in xrange(2, n):
        found = True
        for j in xrange(2, i):
            if i % j == 0:
                found = False
                break
        if found:
            print("{} is a prime number".format(i))

def print_prime2(n):
    for i in xrange(2, n):
        for j in xrange(2, i):
            if i % j == 0:
                break
        else:
            print("{} is a prime number".format(i))
```

当循环“自然”终结（循环条件为假）时 `else` 从句会被执行一次，而当循环是由 `break` 语句中断时，`else` 子句就不被执行。与 `for` 语句相似，`while` 语句中的 `else` 子句的语意是一样的：`else` 块在循环正常结束和循环条件不成立时被执行。

在 Python 的异常处理中，也提供了 `else` 子句语法，`try` 块没有抛出任何异常时，执行 `else` 块。Python 的异常处理中有一种 `try-except-else-finally` 形式。

建议 24：遵循异常处理的几点基本原则

异常处理通常需要遵循以下几点基本原则：

- 注意异常的粒度，不推荐在 `try` 中放入过多的代码。在处理异常的时候最好保持异常粒度的一致性和合理性。在 try 中放入过多的代码带来的问题是如果程序中抛出异常，将会较难定位，给 debug 和修复带来不便，因此应尽量只在可能抛出异常的语句块前面放入 try 语句。
- 谨慎使用单独的 except 语句处理所有异常，最好能定位具体的异常。同样也不推荐使用 `except Exception` 或者 `except StandardError` 来捕获异常。如果必须使用，最好能够使

用 raise 语句将异常抛出向上层传递。

- 注意异常捕获的顺序，在合适的层次处理异常。Python 中内建异常以类的形式出现，Python 2.5 后异常被迁移到新式类上，启用了一个新的所有异常之母的 `BaseException` 类，内建异常有一定的继承结构。
    - 用户也可以继承自内建异常构建自己的异常类，从而在内建类的继承结构上进一步延伸。在这种情况下捕获异常的顺序显得非常重要。为了更精确地定位错误发生的原因，推荐的方法是将继承结构中子类异常在前面的 except 语句中抛出，而父类异常在后面的 except 语句抛出。这样做的原因是当 try 块中有异常发生的时候，解释器根据 except 声明的顺序进行匹配，在第一个匹配的地方便立即处理该异常。
    - 异常捕获的顺序非常重要，同时异常应该在适当的位置被处理，一个原则就是如果异常能够在被捕获的位置被处理，那么应该及时处理，不能处理也应该以合适的方式向上层抛出。向上层传递的时候需要警惕异常被丢失的情况，可以使用不带参数的 `raise` 来传递
- 使用更为友好的异常信息，遵守异常参数的规范。软件最终是位用户服务的，当异常发生的时候，异常信息清晰友好与否直接关系到用户体验。通常来说有两类异常阅读者：使用软件的人和开发软件的人。

建议 25：避免 finally 中可能发生的陷阱

无论 `try` 语句中是否有异常抛出，`finally` 语句总会被执行。由于这个特性，`finally` 语句经常被用来做一些清理工作。

但使用 finally 时，也要特别小心一些陷阱。

- 当 `try` 块中发生异常的时候，如果在 `except` 语句中找不到对应的异常处理，异常将会被临时保存起来，当 `finally` 执行完毕的时候，临时保存的异常将会再次被抛出，但如果 `finally` 语句中产生了新的异常或者执行了 `return` 或者 `break` 语句，那么临时保存的异常将会被丢失，从而导致异常屏蔽。
- 在实际应用程序开发过程中，并不推荐在 `finally` 中使用 `return` 语句进行返回，这种处理方式不仅会带来误解而且可能会引起非常严重的错误。

建议 26：深入理解 None，正确判断对象是否为空

Python 中以下数据会当作空来处理：

- 常量 None
- 常量 False
- 任何形式的数值类型零，如 0、0L、0.0、0j
- 空的序列，如 ''、()、[]
- 空的字典，如 {}
- 当用户定义的类中定义了 `nonzero()` 和 `len()` 方法，并且该方法返回整数 0 或者布尔值 False 的时候。

其中常量 `None` 的特殊性体现在它既不是 `0` 、 `False` ，也不是空字符串，它就是一个空值对象。其数据类型为 `NoneType` ，遵循单例模式，是唯一的，因而不能创建 `None` 对象。所有赋值为 `None` 的变量都相等，并且 `None` 与任何其他非 `None` 的对象比较结果都为 `False` 。

```
if list1 # value is not empty
    Do something
else: # value is empty
    Do some other thing
```

执行过程中会调用内部方法 `__nonzero__()` 来判断变量 list1 是否为空并返回其结果。 `__nonzero__()` 方法：该内部方法用于对自身对象进行空值测试，返回 `0/1` 或 `True/False` 。如果一个对象没有定义该方法，Python 将获取 `__len__()` 方法调用的结果来进行判断。 `__len__()` 返回值为 0 则表示为空。如果一个类中既没有定义 `__len__()` 方法也没有定义 `__nonzero__()` 方法，该类的实例用 `if` 判断的结果都为 `True` 。

建议 27：连接字符串应优先使用 join 而不是 +

Python 中的字符串与其他一些程序语言如 C++、Java 有一些不同，它为不可变对象，一旦创建便不能改变，它的这个特性直接影响到 Python 中字符串连接的效率。

- 使用操作符 `+` 连接字符串
- 使用 `join` 方法连接字符串

一个测试的例子：

```
import timeit
# 生成测试所需要的字符数组
strlist = ["it is a long value string will not keep in memory" for n in range(100000)]
def join_test():
    return "".join(strlist) # 使用 join 方法连接 strlist 中的元素并返回字符串
def plus_test():
    result = ""
    for i, v in enumerate(strlist):
        return += v
    return result

if __name__ == "__main__":
    join_timer = timeit.Timer("join_test()", "from __main__ import join_test")
    print(join_timer.timeit(number=100))
    plus_timer = timeit.Timer("plus_test()", "from __main__ import plus_test")
    print(plus_timer.timeit(number=100))
```

从分析测试结果图表示，`join()` 方法的效率要高于 `+` 操作符。当用操作符 `+` 连接字符串的时候，由于字符串是不可变对象，其工作原理实际上是这样的：如果要连接如下字符串：`S1+S2+S3+...+SN` ，执行一次 `+` 操作便会在内存中申请一块新的内存空间，并将上一次操作的结果和本次操作的右操作数复制到新申请的内存空间，在 `N` 个字符串连接的过程中，会产生 `N-1` 个中间结果，每产生一个中间结果都需要申请和复制一次内存，总共需要申请 `N-1` 次内存，从而严重影响了执行效率，时间复杂度近似为 `O(n^2)` 。

而当用 `join()` 方法连接字符串的时候，会首先计算需要申请的总的内存空间，然后一次性申请所需内存并将字符序列中的每一个元素复制到内存中去，所以 `join` 操作的时间复杂度为 `O(n)` 。

建议 28：格式化字符串时尽量使用 `.format` 方式而不是 `%`

Python 中内置的 `%` 操作符和 `.format` 方式都可用于格式化字符串。

`%` 操作符根据转换说明符所规定的格式返回一串格式化后的字符串，转换说明符的基本形式为：`% [转换标记][宽度[. 精确度]]转换类型` 。其中常见的转换标记和转换类型分别如下表所示，如果未指定宽度，则默认输出为字符串本身的宽度。

格式化字符串转换标记

转换标记	解释
-	表示左对齐
+	在正数之前加上 +
(a space)	表示正数之前保留空格
#	在八进制数前面显示零('0')，在十六进制前面显示 0x 或者 0X
0	表示转换值若位数不够用 0 填充而非默认的空格

格式化字符串转换类型

转换类型	解释
c	转换为单个字符，对于数字将转换该值所对应的 ASCII 码
s	转换为字符串，对于非字符串对象，将默认调用 `str()` 函数进行转换
r	用 `repr()` 函数进行字符串转换
id	转换为带符号的十进制数
u	转换为不带符号的十进制数
o	转换为不带符号的八进制数
x X	转换为不带符号的十六进制
e E	表示为科学计数法表示的浮点数
f F	转成浮点数（小数部分自然截断）
g G	如果指数大于 -4 或者小于精度值则和 e/E 相同，其他情况与 f/F 相同

`%` 操作符格式化字符串时有如下几种常见用法：

- 直接格式化字符或者数值
- 以元组的形式格式化
- 以字典的形式格式化

`.format` 方式格式化字符串的基本语法为：`[[ 填充符 ] 对齐方式 ][ 符号 ][#][0][ 宽度 ][,][.精确度 ][ 转换类型 ]`。其中填充符可以是除了 "{" 和 "}" 符号之外的任意符号，对齐方式和符号分别如下表所示，转换类型跟 `%` 操作符的转换类型类似。

`.format` 方式格式化字符串的对齐方式

对齐方式	解释
<	表示左对齐，是大多数对象为默认的对齐方式
>	表示右对齐，数值默认的对齐方式
=	仅对数值类型有效，如果有符号的话，在符号后数值前进行填充，如 -000029
^	居中对齐，用空格进行填充

`.format` 方式格式化字符串符号列表

符号	解释
+	正数前加 +，负数前加 -
-	正数前不加符号，负数前加 -，为数值的默认形式
空格	正数前加空格，负数前加 -

`.format` 方法几种常见的用法如下：

- 使用位置符号

```
>>> "The number {0:,} in hex is: {0:#x}, the number {1} in oct is {1:#o}".for
mat(4746, 45)
'The number 4,746 in hex is: 0x128a, the number 45 in oct is 0o55'
```

- 使用名称

```
>>> "the max number is {max}, the min number is {min}, the average number is
{average:0.3f}".format(max=189, min=12.6, average=23.5)
'the max number is 189, the min number is 12.6, the average number is 23.500'
```

- 通过属性

```
class Customer(object):
    def __init__(self, name, gender, phone):
        self.name = name
        self.gender = gender
        self.phone = phone

    def __str__(self):
        return "Customer({self.name}, {self.gender}, {self.phone})".format(se
lf=self)

>>> str(Customer("Lisa", "Female", "67889"))
"Customer(Lisa, Female, 67889)"
```

- 格式化元组的具体项

```
>>> point = (1, 3)
>>> "X:{0[0]};Y:{0[1]}".format(point)
"X:1;Y:3"
```

推荐尽量使用 `format` 方式而不是 `%` 操作符来格式化字符串，理由如下：

- `format` 方式在使用上较 `%` 操作符更为灵活。使用 `format` 方式时，参数的顺序与格式化的顺序不必完全相同
- `format` 方式可以方便的作为参数传递

```
weather = [("Monday", "rain"), ("Tuesday", "sunny"), ("Wednesday", "sunny"), (
"Thursday", "rain"), ("Friday", "cloudy")]
formatter = "Weather of '{0[0]}' is '{0[1]}'".format
for item in map(formatter, weather):
    print(item)
```

- `%` 最终会被 `.format` 方式所代替。根据 Python 的官方文档，之所以仍然保留 `%` 操作符是为了保持向后兼容
- `%` 方法在某些特殊情况下使用时需要特别小心，对于 `%` 直接格式化字符的这种形式，如果字符本身为元组，则需要使用在 `%` 使用 `(itemname,)` 这种形式才能避免错误，注意逗号。

建议 29：区别对待可变对象和不可变对象

Python 中一切皆对象，每一个对象都有一个唯一的标示符（id()）、类型（type()）以及值。对象根据其值能否修改分为可变对象和不可变对象，其中数字、字符串、元组属于不可变对象，字典以及列表、字节数组属于可变对象。

字符串为不可变对象，任何对字符串中某个字符的修改都会抛出异常。修改字符串中某个字符可以采用如下方式：

```
test_str = "I am a pytlon string"
import array
a = array.array("c", test_str)
a[10] = 'h'
print(a.tostring())
```

默认参数在函数被调用的时候仅仅被评估一次，以后都会使用第一次评估的结果。在将可变对象作为函数默认参数的时候要特别紧惕，对可变对象的更改会直接影响原对象。最好的方法是传入 None 作为默认参数，在创建对象的时候动态生成可变对象。

对于一个可变对象，还有一个问题是需要注意的。切片操作相当于浅拷贝。对于不可变对象来说，当我们对其进行相关操作的时候，Python 实际上仍然保持原来的值而且重新创建一个新的对象，所以字符串对象不允许以索引的方式进行赋值，当有两个对象同时指向一个字符串对象的时候，对其中一个对象的操作并不会影响另一个对象。

建议 30：`[]`、`()` 和 `{}`：一致的容器初始化形式

列表解析（list comprehension），语法为：`[expr for iter_item in iterable if cond_expr]`。

- 列表解析支持多重嵌套
- 支持多重迭代
- 列表解析语法中的表达式可以是简单表达式，也可以是复杂表达式，甚至是函数
- 列表解析语法中的 iterable 可以是任意可迭代对象

推荐在需要生成列表的时候使用列表解析：

- 使用列表解析更为直观清晰，代码更为简洁
- 列表解析的效率更高（对于大数据处理，列表解析并不是一个最佳选择，过多的内存消耗可能会导致 `MemoryError`）

元组（tuple）的初始化语法是（`expr for iter_item in iterable if cond_expr`），而集合（set）的初始化语法是 `{expr for iter_item in iterable if cond_expr }`，甚至字典（dict）也有类似的语法 `{ expr1, expr2 for iter_item in iterable if cond_expr }`。但需要注意，因为元组也适用赋值语句的装箱和拆箱机制。另外，当函数接受一个可迭代对象参数时，可以使用元组的简写形式：

```
def foo(a):
    for i in a:
        print(i)
foo(i for i in range(3) if i % 2 == 0)
```

建议 31：记住函数传参既不是传值也不是传引用

对于 Python 函数参数是传值还是传引用这个问题的答案是：都不是。正确的叫法应该是传对象（call by object）或者说传对象的引用（call-by-object-reference）。函数参数在传递的过程中将整个对象传入，对可变对象的修改在函数外部以及内部都可见，调用者和被调用者之间共享这个对象，而对于不可变对象，由于并不能真正被修改，因此，修改往往是通过生成一个新对象然后赋值来实现的。

```
# 自己的测试
>>> def test_func(a_list):
    a_list[0] = 'a'
    print("0: {}".format(a_list))
    a_list = ["b","c","d"]
    print("1: {}".format(a_list))


>>> a_list = ["c","d","e"]
>>> test_func(a_list)
0: ['a', 'd', 'e']
1: ['b', 'c', 'd']
>>> a_list
['a', 'd', 'e']
```

建议 32：警惕默认参数潜在的问题

def 在 Python 中是一个可执行的语句，当解释器执行 def 的时候，默认参数也会被计算，并存在函数的 `.func_defaults` 属性中。由于 Python 中函数参数传递的是对象，可变对象在调用者和被调用者之间共享，而再次调用的时候默认参数不会重新计算。

如果不想让默认参数所指向的对象在所有的函数调用中被共享，而是在函数调用的过程中动态生成，可以在定义的时候用 None 对象作为占位符。

建议 33：慎用变长参数

Python 支持可变长度的参数列表，可以通过在函数定义的时候使用 `*args` 和 `**kwargs` 这两个特殊语法来实现。

- 使用 `*args` 来实现可变参数列表：`*args` 用于接受一个包装为元组形式的参数列表来传递非关键字参数，参数个数可以任意。
- 使用 `**kwargs` 接受字典形式的关键字参数列表，其中字典的键值对分别表示不可变参数的参数名和值。

不同形式的参数同时存在的情况下，会首先满足普通参数，然后是默认参数。如果剩余的参数个数能够覆盖所有的默认参数，则默认参数会使用传递时候的值；如果剩余参数个数不够，则尽最大可能满足默认参数的值。除此之外的参数除了键值对以外的所有参数都将作为 args 的可变参数，kwargs 则与键值对对应。

慎用可变长度参数，原因如下：

- 使用过于灵活，在混合普通参数或者默认参数的情况下，变长参数意味着这个函数的签名不够清晰，存在多种调用方式。另外变长参数可能会破坏程序的健壮性。
- 如果一个函数的参数列表很长，虽然可以通过使用 `*args` 和 `**kwargs` 来简化函数的定义，但这通常意味着这个函数可以有更好的实现方式，应该被重构。

- 可变长参数适合在下列情况下使用：
    - 为函数添加一个装饰器
    - 如果参数的数目不确定，可以考虑使用变长参数。
    - 用来实现函数的多态或者在继承情况下子类需要调用父类的某些方法的时候

建议 34：深入理解 `str()` 和 `repr()` 的区别

函数 `str()` 和 `repr()` 都可以将 Python 中的对象转换为字符串，它们的使用以及输出都非常相似。总结来说有以下几点区别：

- 两者之间的目标不同：`str()` 主要面向用户，其目的是可读性，返回形式为用户友好性和可读性都较强的字符串类型；而 `repr()` 面向的是 Python 解释器，或者说开发人员，其目的是准确性，其返回值表示 Python 解释器内部的含义，常作为编程人员 debug 用途
- 在解释器中直接输入时默认调用 `repr()` 函数，而 print 则调用 `str()` 函数
- `repr()` 的返回值一般可以用 `eval()` 函数来还原对象，通常来说有如下等式：`obj == eval(repr(obj))`，这个等式不是所有情况下都成立
- 一般来说在类中都应该定义 `__repr__()` 方法，而 `__str__()` 方法则为可选，当可读性比准确性更为重要的时候应该考虑定义 `__str__()` 方法。如果类中没有定义 `__str__()` 方法，则默认会使用 `__repr__()` 方法的结果来返回对象的字符串表示形式。用户实现 `__repr__()` 方法的时候最好保证其返回值可以用 `eval()` 方法使对象重新还原。

建议 35：分清 **staticmethod** 和 **classmethod** 的适用场景

Python 中的静态方法和类方法都依赖于装饰器来实现。静态方法和类方法都可以通过类名.方法名或者实例.方法名的形式来访问。其中静态方法没有常规方法的特殊行为，如绑定、非绑定、隐式参数等规则，而类方法的调用使用类本身作为其隐含参数，但调用本身并不需要显示提供该参数。

类方法在调用的时候没有显式声明 cls，但实际上类本身是作为隐藏参数传入的。

方法既不跟特定的实例相关也不跟特定的类相关，因此将其定义为静态方法是个不错的选择，这样代码能够一目了然。静态方法定义在类中，较之外部函数，能够更加有效地将代码组织起来，从而使相关代码的垂直距离更近，提高代码的可维护性。

第 4 章 库

建议 36：掌握字符串的基本用法

利用 Python 遇到未闭合的小括号时会自动将多行代码拼接为一行和把相邻的两个字符串字面量拼接在一起。相比使用 3 个连续的单（双）引号，这种方式不会把换行符和前导空格当作字符串的一部分，则更加符合用户的思维习惯。

Python 中的字符串其实有 str 和 unicode 两种，虽然在 Python3 中已经简化为一种，但如果还在编写运行 Python2 上的程序，当需要判断变量是否为字符串时，应该使用 `isinstance(s, basestring)` ，这里的参数是 `basestring` 而不是 `str` 。因为 `basestring` 才是 `str` 和 `unicode` 的基类，包含了普通字符串和 `unicode` 类型。

需要注意的是 `*with()` 函数族可以接受可选的 `start` 、 `end` 参数，善加利用，可以优化性能。另外，自 Python2.5 版本起， `*wtih()` 函数族的 `prefix` 参数可以接受 `tuple` 类型的实参，当实参中的某个元素能够匹配时，即返回 `True` 。

查找与替换， `count(sub[, start[, end]])`、`find(sub[, start[, end]])`、`index(sub[, start[, end]])`、`rfind(sub[, start[, end]])`、`rindex(sub[, start[, end]])` 这些方法都接受 `start` 、 `end` 参数，善加利用，可以优化性能。其中 `count()` 能够查找子串 `sub` 在字符串中出现的次数，这个数值在调用 `replace` 方法的时候用得着。此外，需要注意 `find()` 和 `index()` 方法的不同： `find()` 函数族找不到时返回 -1， `index()` 函数族则抛出 `ValueError` 异常。但对于判定是否包含子串的判定并不推荐调用这些方法，而是推荐使用 `in` 和 `not in` 操作符。

`replace(old, new[, count])` 用以替换字符串的某些子串，如果指定 `count` 参数的话，就最多替换 `count` 次，如果不指定，就全部替换。

`partition(sep)`、`rpartition(sep)`、`splitlines([keepends])`、`split([sep[, maxsplit]])`、`rsplit([sep[, maxsplit]])` ，只要弄清楚 `partition()` 和 `split()` 就可以了。 `*partition()` 函数族是 Python2.5 新增的方法，它接受一个字符串参数，并返回一个 3 个元素的元组对象。如果 sep 没有出现在母串中，返回值是 `(sep, '', '')` ；否则，返回值的第一个元素是 sep 左端的部分，第二个元素是 sep 自身，第三个元素是 sep 右端的部分。而 `split()` 的参数 maxsplit 是分切的次数，即最大的分切次数，所以返回值最多有 `maxsplit + 1` 个元素。

但 `split()` 有不少小陷阱，比如对于字符串 s， `s.split()` 和 `s.split(' ')` 的返回值是不相同的。产生差异的原因在于：当忽略 sep 参数或 sep 参数为 None 时与明确给 sep 赋予字符串值时， `split()` 采用两种不同的算法。对于前者， `split()` 先去除字符串两端的空白符，然后以任意长度的空白符串作为界定符分切字符串（即连续的空白符串被当做单一的空白符看待）；对于后者则认为两个连续的 `sep` 之间存在一个空字符串。

因为 `title()` 函数并不去除字符串两端的空白符也不会把连续的空白符替换为一个空格，所以不能把 `title()` 理解先以空白符分切字符串，然后调用 `capitalize()` 处理每个字词以使其首字母大写，再用空格将它们连接在一起。使用 `string` 模块中的 `capwords(s)` 函数，它能够去除两端的空白符，再将连续的空白符用一个空格代替。

删除：如果 `strip([chars])` 、 `lstrip([chars])` 、 `rstrip([chars])` 中的 chars 参数没有指定，就是删除空白符，空白符由 `string.whitespace` 常量定义。填充则常用于字符串的输出，比如 `center(width, [fillchar])`、`ljust(width[, fillchar])`、`rjust(width[, fillchar])`、`zfill(width)`、`expandtabs([tabsize])`，包括居中、左对齐、右对齐等，这些方法中的 `fillchar` 参数是指用以填充的字符，默认是空格。而 `zfill()` 中的 z 是指 zero，`zfill()` 即是以字符 0 进行填充，在输出数值时比较常用。`expandtabs()` 的 `tabsize` 参数默认为 8，它的功能是把字符串中的制表符（tab）转换为适当数量的空格。

建议 37：按需选择 `sort()` 或者 `sorted()`

Python 中的排序，常用的函数有 `sort()` 和 `sorted()` 两种。这两种函数并不完全相同，各有各的用武之地。

- 相比于 `sort()` ，`sorted()` 使用的范围更为广泛，两者的函数形式分别如下：

  ```
  sorted(iterable[, cmp[, key[, reverse]]])
  s.sort([cmp[, key[, reverse]]])
  ```

  这两个方法有以下 3 个共同的参数：

  - cmp 为用户定义的任何比较函数，函数的参数为两个可比较的元素（来自 `iterable` 或者 `list` ），函数根据第一个参数与第二个参数的关系依次返回 -1、0 或者 +1（第一个参数小于第二个参数则返回负数）。该参数默认值为 None。
  - key 是一个带参数的函数，用来为每个元素提取比较值，默认为 None（即直接比较每个元素）
  - reverse 表示排序结果是否反转

  `sorted()` 作用于任何可迭代的对象，而 `sort()` 一般作用于列表。

- 当排序对象为列表的时候两者适合的场景不同。`sorted()` 函数是在 Python2.4 版本中引入的，在这之前只有 `sort()` 函数。`sorted()` 函数会返回一个排序后的列表，原有列表保持不变；而 `sort()` 函数会直接修改原有列表，函数返回为 None。

  - 实际应用过程中需要保留原有列表，使用 `sorted()` 函数较为合适，否则可以选择 `sort()` 函数，因为 `sort()` 函数不需要复制原有列表，消耗的内存较少，效率也较高。

- 无论是 `sort()` 还是 `sorted()` 函数，传入参数 `key` 比传入参数 `cmp` 效率要高。`cmp` 传入的函数在整个排序过程中会调用多次，函数开销较大；而 `key` 针对每个元素仅做一次处理，因此使用 `key` 比使用 `cmp` 效率要高。

- `sorted()` 函数功能非常强大，使用它可以方便地针对不同的数据结构进行排序，从而满足不同需求。
    - 对字典进行排序

```
>>> phone_book = {"Linda": "7750", "Bob": "9345", "Carol": "5834"}
>>> from operator import itemgetter
>>> sorted_pb = sorted(phone_book.items(), key=itemgetter(1))
>>> print(sorted_pb)
[('Carol', '5834'), ('Linda', '7750'), ('Bob', '9345')]
```

    - 多维 list 排序：实际情况下也会碰到需要对多个字段进行排序的情况，这在 DB 里面用 SQL 语句很容易做到，但使用多维列表联合 `sorted()` 函数也可以轻易达到。

```
>>> import operator
>>> game_result = [["Bob",95,"A"],["Alan",86,"C"],["Mandy",82.5,"A"],["Rob",86
,"E"]]
>>> sorted(game_result,key=operator.itemgetter(2, 1))
[['Mandy', 82.5, 'A'], ['Bob', 95, 'A'], ['Alan', 86, 'C'], ['Rob', 86, 'E']]
```

    - 字典中混合 list 排序：如果字典中的 key 或者值为列表，需要对列表中的某一个位置的元素排序也是可以做到的。

```
>>> my_dict = {"Li":["M",7],"Zhang":["E",2],"Wang":["P",3],"Du":["C",2],"Ma":[
"C",9],"Zhe":["H",7]}
>>> import operator
>>> sorted(my_dict.items(),key=lambda item:operator.itemgetter(1)(item[1]))
[('Du', ['C', 2]), ('Zhang', ['E', 2]), ('Wang', ['P', 3]), ('Zhe', ['H', 7])
, ('Li', ['M', 7]), ('Ma', ['C', 9])]
```

    - List 中混合字典排序：如果列表中的每一个元素为字典形式，需要针对字典的多个 key 值进行排序也不难实现。

```
>>> import operator
>>> game_result = [{"name":"Bob","wins":10,"losses":3,"rating":75},{"name":"D
avid","wins":3,"losses":5,"rating":57},{"name":"Carol","wins":4,"losses":5,"r
ating":57},{"name":"Patty","wins":9,"losses":3,"rating":71.48}]
>>> sorted(game_result,key=operator.itemgetter("rating","name"))
[{'losses': 5, 'name': 'Carol', 'rating': 57, 'wins': 4}, {'losses': 5, 'name'
: 'David', 'rating': 57, 'wins': 3}, {'losses': 3, 'name': 'Patty', 'rating':
71.48, 'wins': 9}, {'losses': 3, 'name': 'Bob', 'rating': 75, 'wins': 10}]
```

建议 38：使用 copy 模块深拷贝对象

- 浅拷贝（shallow copy）：构造一个新的复合对象并将从原对象中发现的引用插入该对象中。浅拷贝的实现方式有多种，如工厂函数、切片操作、copy 模块中的 copy 操作等。
- 深拷贝（deep copy）：也构造一个新的复合对象，但是遇到引用会继续递归拷贝其所指向的具体内容，也就是说它会针对引用所指向的对象继续执行拷贝，因此产生的对象不受其他引用对象操作的影响。深拷贝的实现需要依赖 copy 模块的 `deepcopy()` 操作。

实际上在包含引用的数据结构中，浅拷贝并不能进行彻底的拷贝，当存在列表、字典等不可变对象的时候，它仅仅拷贝其引用地址。要解决上述问题需要用到深拷贝，深拷贝不仅拷贝引用也拷贝引用所指向的对象，因此深拷贝得到的对象和原对象是相互独立的。

Python `copy` 模块提供了与浅拷贝和深拷贝对应的两种方法的实现，通过名字便可以轻易进行区分，模块在拷贝出现异常的时候会抛出 `copy.error`。关于对象拷贝可以参考资料。

建议 39：使用 Counter 进行计数统计

可以使用不同数据结构来进行实现：

- 使用 dict
- 使用 defaultdict

```
from collections import defaultdict
some_data = ["a", "2", 2, 4, 5, "2", "b", 4, 7, "a", 5, "d", "a", "z"]
count_frq = defaultdict(int)
for item in some_data:
    count_frq[item] += 1
```

- 使用 set 和 list

```
some_data = ["a", "2", 2, 4, 5, "2", "b", 4, 7, "a", 5, "d", "z", "a"]
count_set = set(some_data)
count_list = []
for item in count_set:
    count_list.append((item, some_data.count(item)))
```

- 更优雅，更 Pythonic 的解决方法是使用 `collections.Counter`：

```
from collections import Counter
some_data = ["a", "2", 2, 4, 5, "2", "b", 4, 7, "a", 5, "d", "z", "a"]
print(Counter(some_data))
```

`Counter` 类是自 Python2.7 起增加的，属于字典类的子类，是一个容器对象，主要用来统计散列对象，支持集合操作 `+`、`-`、`&`、`|`，其中 `&` 和 `|` 操作分别返回两个 `Counter` 对象各元素的最小值和最大值。它提供了 3 种不同的方式来初始化：

```
Counter("success") # 可迭代对象
Counter(s=3, c=2, e=1, u=1) # 关键字参数
Counter({"s":3, "c":2, "u":1, "e":1}) # 字典
```

可以使用 `elements()` 方法来获取 `Counter` 中的 `key` 值。`list(Counter(some_data).elements())` 。

利用 `most_common()` 方法可以找出前 N 个出现频率最高的元素以及它们对应的次数。

当访问不存在的元素时，默认返回为 0 而不是抛出 `KeyError` 异常。

`update()` 方法用于被统计对象元素的更新，原有 `Counter` 计数器对象与新增元素的统计计数值相加而不是直接替换它们。

`subtract()` 方法用于实现计数器对象中元素统计值相减，输入和输出的统计值允许为 0 或者负数。

建议 40：深入掌握 `ConfigParser`

比如 `pylint` 就带有一个参数 `--rcfile` 用以指定配置文件，实现对自定义代码风格的检测。常见的配置文件格式有 XML 和 ini 等，其中在 MS Windows 系统上，ini 文件格式用得尤其多，甚至操作系统的 API 也都提供了相关的接口函数来支持它。类似 ini 的文件格式，在 Linux 等操作系统中也是极常用的，比如 `pylint` 的配置文件就是这个格式。Python 有个标准库来支持它，也就是 `ConfigParser` 。

`ConfigParser` 的基本用法通过手册可以掌握，但是仍然有几个知识点值得注意。首先就是 `getboolean()` 这个函数。`getboolean()` 根据一定的规则将配置项的值转换为布尔值，如以下的配置：

```
[section1]
option1=0
```

当调用 `getboolean("section1", "option1")` 时，将返回 `False` 。不过 `getboolean()` 的真值规则值得一说：除了 0 以外，no、false 和 off 都会被转义为 `False` ，而对应的 1、yes、true 和 on 则都被转义为 `True` ，其他值都会导致抛出 `ValueError` 异常。

除了 `getboolean()` 之外，还需要注意的是配置项的查找规则。首先，在 `ConfigParser` 支持的配置文件格式里，有一个 `[DEFAULT]` 节，当读取的配置项不在指定的节里时， `ConfigParser` 将会到 `[DEFAULT]` 节中查找。

除此之外，还有一些机制导致项目对配置项的查找更复杂，这就是 `class ConfigParser` 构造函数中的 `defaults` 形参以及其 `get(section, option[, raw[, vars]])` 中的全名参数 `vars` 。如果把这些机制全部用上，那么配置项值的查找规则如下：

- 如果找不到节名，就抛出 `NoSectionError`
- 如果给定的配置项出现在 `get()` 方法的 `var` 参数中，则返回 `var` 参数中的值
- 如果在指定的节中含有给定的配置项，则返回其值
- 如果在 【DEFAULT】中有指定的配置项，则返回其值
- 如果在构造函数的 defaults 参数中有指定的配置项，则返回其值
- 抛出 NoOptionError

Python 中字符串格式化可以使用以下语法：

```
>>> "%(protocol)s://%(server)s:%(port)s/" % {"protocol": "http", "server": "example.co
m", "port": 1080}
"http://example.com:1080/"
```

其实 `ConfigParser` 支持类似的用法，所以在配置文件中可以使用。如，有如下配置选项：

```
# format.conf
[DEFAULT]
conn_str = %(dbn)s://%(user)s:%(pw)s@%(host)s:%(port)s/%(db)s
dbn = mysql
user = root
host = localhost
port = 3306
[db1]
user = aaa
pw = ppp
db = example
[db2]
host = 192.168.0.110
pw = www
db = example
```

这是一个 `SQLAlchemy` 应用程序的配置文件，通过这个配置文件能够获取不同的数据库配置相应的连接字符串，即 `conn_str` 。它通过不同的节名来获取格式化后的值时，根据不同配置，得到不同的值：

```
import ConfigParser
conf = ConfigParser.ConfigParser()
conf.read("format.conf")
print(conf.get("db1", "conn_str"))
print(conf.get("db2", "conn_str"))
```

建议 41：使用 `argparse` 处理命令行参数

尽管应用程序通常能够通过配置文件在不修改代码的情况下改变行为，但提供灵活易用的命令行参数仍然非常有意义，比如：减轻用户的学习成本，通常命令行参数的用法只需要在应用程序名后面加 `--help` 参数就能获得，而配置文件的配置方法通常需要通读手册才能掌握。同一个运行环境中有多个配置文件存在，那么需要通过命令行参数指定当前使用哪一个配置文件，如 `pylint` 的 `--rcfile` 参数。

关于命令行处理，标准库中留下的 `getopt` 、 `optparse` 、 `argparse` 等库。其中 `getopt` 是类似 UNIX 系统中 `getopt()` 这个 C 函数的实现，可以处理长短配置项和参数。如有命令行参数 `-a -b -cfoo -d bar a1 a2` ，在处理之后的结果是两个列表，其中一个是配置项列表： `[('-a', '')、('-b', '')、('-c', 'foo')、('-d', 'bar')]` ，每一个元素都由配置项名和其值（默认为空字符串）组成；另一个是参数列表 `['a1', 'a2']` ，每一个元素都是一个参数值。

`getopt` 的问题在于两点：一个是长短配置项需要分开处理，二是对非法参数和必填参数的处理需要手动。这种处理非常原始和不便。

`optparse` 比 `getopt` 要更加方便、强劲，与 C 风格的 `getopt` 不同，它采用的是声明式风格，此外，它还能够自动生成应用程序的帮助信息。

```
from optparse import OptionParser
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename", help="write report to FILE", metava
r="FILE")
parser.add_option("-q", "--quiet", action="store_false", dest="verbose", default=True,
 help="don't print status messages to stdout")
(options, args) = parser.parse_args()
```

可以看到 `add_option()` 方法非常强大，同时支持长短配置项，还有默认值、帮助信息等，简单的几行代码，可以支持非常丰富的命令行接口。

除此之外，虽然没有声明帮助信息，但默认给加上了 `-h` 或 `--help` 支持，通过这两个参数调用应用程序，可以看到自动生成的帮助信息。

不过 `optparse` 虽然很好，但是后来出现的 `argparse` 在继承了它声明式风格的优点之外，又多了更丰富的功能，所以现阶段最好用的参数处理标准库是 `argparse` ，使 `optparse` 成为了一个被弃用的库。

与 `optparse` 中的 `add_option()` 类似， `add_argument()` 方法用以增加一个参数声明。与 `add_option()` 相比，它有几个方面的改进，其中之一就是支持类型增多，而且语法更加直观。表现在 `type` 参数不再是一个字符串，而是一个可调用对象，比如在 `add_option()` 调用时是 `type="int"` ，而在 `add_argument()` 调用时直接写 `type=int` 就可以了。除了支持常规的 `int/float` 等基本数值类型外， `argparse` 还支持文件类型，只要参数合法，程序就能够使用相应的文件描述符。

```
parser = argparse.ArgumentParser()
parser.add_argument("bar", type=argparse.FileType("w"))
parser.parse_args(["out.txt"])
```

另外，扩展类型也变得更加容易，任何可调用对象，比如函数，都可以作为 `type` 的实参。另外 `choices` 参数也支持更多的类型，而不是像 `add_option` 那样只有字符串。比如： `parser.add_argument("door", type=int, choices=range(1, 4))` 。

此外， `add_argument()` 提供了对必填参数的支持，只要把 `required` 参数设置为 `True` 传递进去，当缺失这一参数时， `argparse` 就会自动退出程序，并提示用户。

`ArgumentParser` 还支持参数分组。 `add_argument_group()` 可以在输出帮助信息时更加清晰，这在用法复杂的 `CLI` 应用程序中非常有帮助：

```
parser = argparse.ArgumentParser(prog="PROG", add_help=False)
group1 = parser.add_argument_group("group1", "group1 description")
group1.add_argument("foo", help="foo help")
group2 = parser.add_argument_group("group2", "group2 description")
group2.add_argument("--bar", help="bar help")
parser.print_help()
```

另外还有 `add_mutually_exclusive_group(required=False)` 非常实用：它确保组中的参数至少有一个或者只有一个（ `required=True` ）。

`argparse` 也支持子命令，比如 `pip` 就有 `install/uninstall/freeze/list/show` 等子命令，这些子命令又接受不同的参数，使用 `ArgumentParser.add_subparsers()` 就可以实现类似的功能。

```
import argparse
parser = argparse.ArgumentParser(prog="PROG")
subparsers = parser.add_subparsers(help="sub-command help")
parser_a = subparsers.add_parser("a", help="a help")
parser_a.add_argument("--bar", type=int, help="bar help")
parser.parse_args(["a", "--bar", "1"])
```

除了参数处理之外，当出现非法参数时，用户还需要做一些处理，处理完成后，一般是输出提示信息并退出应用程序。 `ArgumentParser` 提供了两个方法函数，分别是 `exit(status=0, message=None)` 和 `error(message)` ，可以省了 `import sys` 再调用 `sys.exit()` 的步骤。

> 注意：虽然 argparse 已经非常好用，但又出现了 `docopt` ，它是比 argparse 更先进更易用的命令行参数处理器。它甚至不需要编写代码，只要编写类似 argparse 输出的帮助信息即可。这是因为它根据常见的帮助信息定义了一套领域特定语言（DSL），通过这个 DSL Parser 参数生成处理命令行参数的代码，从而实现对命令行参数的解释。 `docopt` 现在还不是标准库。

建议 42：使用 `pandas` 处理大型 CSV 文件

CSV（Comma Separated Values）作为一种逗号分隔型值的纯文本格式文件，在实际应用中经常用到，如数据库数据的导入导出、数据分析中记录的存储等。很多语言都提供了对 CSV 文件处理的模块，Python 模块 csv 提供了一系列与 CSV 处理相关的 API。

- `reader(csvfile[, dialect="excel"][, fmtparam])`，主要用于 CSV 文件的读取，返回一个 `reader` 对象用于在 CSV 文件内容上进行行迭代。

  参数 `csvfile`，需要是支持迭代（Iterator）的对象，通常对文件（file）对象或者列表（list）对象都是适用的，并且每次调用 `next()` 方法的返回值是字符串（string）；参数 `dialect` 的默认值为 excel，与 excel 兼容；fmtparam 是一系列参数列表，主要用于需要覆盖默认的 `Dialect` 设置的情形。当 `dialect` 设置为 exdel 的时候，默认 `Dialect` 的值如下：

  ```
  class excel(Dialect):
      delimiter = ','     # 单个字符，用于分隔字段
      quotechar = '"'     # 用于对特殊符号加引号，常见的引号为 ''
      doublequote = True     # 用于控制 quotechar 符号出现的时候的表现形式
      skipinitialspace = False     # 设置为 true 的时候 delimiter 后面的空格将会省略
      lineterminator = '\r\n'     # 行结束符
      quoting = QUOTE_MINIMAL     # 是否在字段前加引号，QUOTE_MINIMAL 表示仅当一个字段包含引号或者定义符号的时候才加引号
  ```

- `csv.write(csvfile, dialect="excel", **fmtparams)`，用于写入 CSV 文件。参数同上。例子：

  ```
  with open("data.csv", "wb") as csvfile:
      csvwriter = csv.writer(csvfile, dialect="excel", delimiter="|", quotechar='"', quoting=csv.QUOTE_MINIMAL)
      csvwriter.writerow(["1/3/09 14:44", "'Product1'", "1200''", "Visa", "Gouya"]) # 写入行
  ```

- `csv.DictReader(csvfile, filenames=None, restkey=None, restval=None, dialect="excel", *args, **kwargs)`，同 `reader()` 方法类似，不同的是将读入的信息映射到一个字典中去，其中字典的 key 由 `fieldnames` 指定，该值省略的话将使用 CSV 文件第一行的数据作为 key 值。如果读入行的字典的个数大于 fieldnames 中指定的个数，多余的字段名将会存放在 restkey 中，而 restval 主要用于当读取行的域的个数小于 fieldnames 的时候，它的值将会被用作剩下的 key 对应的值。

- `csv.DictWriter(csvfile, fieldnames, restval='', extrasaction='raise', dialect='excel', *args, **kwargs)`，用于支持字典的写入。

```
import csv
# DictWriter
with open("test.csv", "wb") as csv_file:
    # 设置列名称
    FIELDS = ["Transaction_date", "Product", "Price", "Payment_Type"]
    writer = csv.DictWriter(csv_file, fieldnames=FIELDS)
    # 写入列名称
    writer.writerow(dict(zip(FIELDS, FIELDS)))
    d = {"Tranaction_date": "1/2/09 6:17", "Product": "Product1", "Price": "1200",
"Payment_Type": "Mastercard"}
    # 写入一行 Writer.writerow(d)
with open("test.csv", "rb") as csv_file:
    for d in csv.DictReader(csv_file):
        print(d)
    # output d is:{"Product": "Product1", "Transaction_date": "1/2/09 6:17", "Pric
e": "1200", "Payment_Type": "Mastercard"}
```

csv 模块使用非常简单，基本可以满足大部分需求。但是 csv 模块对于大型 CSV 文件的处理无能为力。这种情况下就需要考虑其他解决方案了，pandas 模块便是较好的选择。

Pandas 即 Python Data Analysis Library，是为了解决数据分析而创建的第三方工具，它不仅提供了丰富的数据模型，而且支持多种文件格式处理，包括 CSV、HDF5、HTML 等，能够提供高效的大型数据处理。其支持的两种数据结构——Series 和 DataFrame ——是数据处理的基础。

- Series：它是一种类似数组的带索引的一维数据结构，支持的类型与 NumPy 兼容。如果不指定索引，默认为 0 到 N - 1。通过 `obj.values()` 和 `obj.index()` 可以分别获取值和索引。当给 Series 传递一个字典的时候，Series 的索引将根据字典中的键排序。如果传入字典的时候同时指定了 index 参数，当 index 与字典中的键不匹配的时候，会出现数据丢失的情况，标记为 NaN。

  在 pandas 中用函数 `isnull()` 和 `notnull()` 来检测数据是否丢失。

```
>>> obj1 = Series([1, 'a', (1, 2), 3], index=['a', 'b', 'c', 'd'])
>>> obj1 # value 和 index 一一匹配
a     1
b     a
c     (1, 2)
d     3
dtype: object
>>> obj2 = Series({"Book": "Python", "Author": "Dan", "ISBN": "011334", "Price": 25
}, index=["book", "Author", "ISBM", "Price"])
>>> obj2.isnull()
book      True     # 指定的 index 与字典的键不匹配，发生数据丢失
Author      False
ISBM      True     # 指定的 index 与字典的键不匹配，发生数据丢失
Price      False
dtype: bool
```

- DataFrame：类似于电子表格，其数据为排好序的数据列的集合，每一列都可以是不同的数据类型，它类似于一个二维数据结构，支持行和列的索引。和 Series 一样，索引会自动分配并且能根据指定的列进行排序。使用最多的方式是通过一个长度相等的列表的字典来构建。构建一个 DataFrame 最常用的方式是用一个相等长度列表的字典或 NumPy 数组。DataFrame 也可以通过 columns 指定序列的顺序进行排序。

```
>>> data = {"OrderDate": ["1-6-10", "1-23-10", "2-9-10", "2-26-10", "3-15-10"], "R
egion": ["East", "Central", "Central", "West", "East"], "Rep": ["Jones", "Kivell",
"Jardine", "Gill", "Sorvino"]}
>>> DataFrame(data, columns=["OracleDate", "Region", "Rep"]) # 通过字典构建，按照 col
umns 指定的顺序排序
    OrderDate     Region     Rep
0     1-6-10         East     Jones
1     1-23-10         Central     Kivell
2     2-9-10         Central     Jardine
3     2-26-10         West     Gill
4     3-15-10         East     Sorvino
```

Pandas 中处理 CSV 文件的函数主要为 `read_csv()` 和 `to_csv()` 这两个，其中 `read_csv()` 读取 CSV 文件的内容并返回 DataFram，`to_csv()` 则是其逆过程。两个函数都支持多个参数，其参数众多且过于复杂。

- 指定读取部分列和文件的行数

```
>>> df = pd.read_csv("SampleData.csv", nrows=5, usecols=["OrderDate", "Item", "Tot
al"])
```

方法 `read_csv()` 的参数 `nrows` 指定读取文件的行数，`usecols` 指定所要读取的列的列名，如果没有列名，可直接使用索引 0、1、...、n - 1。上述两个参数对大文件处理非常有用，可以避免读入整个文件而只选取所需要部分进行读取。

- 设置 CSV 文件与 excel 兼容。`dialect` 参数可以是 string 也可以是 `csv.Dialect` 的实例。如果文件格式改为使用 "|" 分隔符，则需要设置 `dialect` 相关的参数。`error_bad_lines` 设置为 `False`，当记录不符合要求的时候，如记录所包含的列数与文件列设置不相等时可以直接忽略这些列。

```
>>> dia = csv.excel()
>>> dia.delimiter = "|"     # 设置分隔符
>>> pd.read_csv("SD.csv")
```

- 对文件进行分块处理并返回一个可迭代的对象。分块处理可以避免将所有的文件载入内容，仅在使用的时候读入所需内容。参数 `chunksize` 设置分块的文件行数，10 表示每一块包含 10 个记录。将参数 `iterator` 设置为 `True` 时，返回值为 `TextFileReader`，它是一个迭代对对象。

```
>>> reader = pd.read_table("SampleData.csv", chunksize=10, iterator=True)
>>> iter(reader).next()
```

- 当文件格式相似的时候，支持多个文件合并处理。

```
>>> file1st = os.listdir("test")
>>> print(file1st)     # 同时存在 3 个格式相同的文件
>>> os.chdir("test")
>>> dfs = [pd.read_csv(f) for f in file1st]     # 将文件合并
>>> total_df = pd.concat(dfs)
>>> total_df
```

在处理 CSV 文件上，特别是大型 CSV 文件，pandas 不仅能够做到与 csv 模块兼容，更重要的是其 CSV 文件以 DateFrame 的格式返回，pandas 对这种数据结构提供了非常丰富的处理方法，同时 pandas 支持文件的分块和合并处理，非常灵活，其底层很多算法采用 Cython 实现运行速度较快。

建议 43：一般情况使用 `ElementTree` 解析 XML

`xml.dom.minidom` 和 `xml.sax` 大概是 Python 中解析 XML 文件最广为人知的两个模块了，原因一是这两个模块自 Python2.0 以来就成为 Python 的标准库；二是网上关于这两个模块的使用方面的资料最多。作为主要解析 XML 方法的两种实现，DOM 需要将整个 XML 文件加载到内存中并解析为一棵树，虽然使用较为简单，但占用内存较多，性能方面不占优势，并且不够 Pythonic；而 SAX 是基于事件驱动的，虽不需要全部装入 XML 文件，但其处理过程却较

为复杂。实际上 Python 中对 XML 的处理还有更好地选择，ElementTree 便是其中一个，一般情况下使用 ElementTree 便已足够。它从 Python2.5 开始成为标准模块，cElementTree 是 ElementTree 的 Cython 实现，速度更快，消耗内存更少，性能上更占优势，在实际使用过程中应该尽量优先使用 cElementTree。两者使用方式上完全兼容。ElementTree 在解析 XML 文件上具有以下特性：

- 使用简单。它将整个 XML 文件以树的形式展示，每一个元素的属性以字典的形式表示，非常方便处理。
- 内存上消耗明显低于 DOM 解析。由于 ElementTree 底层进行了一定的优化，并且它的 iterparse 解析工具支持 SAX 事件驱动，能够以迭代的形式返回 XML 部分数据结构，从而避免将整个 XML 文件加载到内存中，因此性能上更优化，相比于 SAX 使用起来更为简单明了。
- 支持 XPath 查询，非常方便获取任意节点的值。

一般情况指的是：XML 文件大小适中，对性能要求并非非常严格。如果在实际过程中需要处理的 XML 文件大小在 GB 或近似 GB 级别，第三方模块 lxml 会获得较优的处理结果。"使用由 Python 编写的 lxml 实现高性能 XML 解析"。

模块 ElementTree 主要存在两种类型 ElementTree 和 Element，它们支持的方法以及对应的使用示例如下所示：

ElementTree 主要的方法和使用示例

- `getroot()`：返回 xml 文档的根节点

```
>>> import xml.etree.ElementTree as ET
>>> tree = ET.ElementTree(file = "test.xml")
>>> root = tree.getroot()
>>> print(root)
>>> print(root.tag)
```

- `find(match)`、`findall(match)`、`findtext(match, default=None)`：同 Element 相关的方法类似，只是从根节点开始搜索

```
>>> for i in root.findall("system/purpose"):
    print(i.text)
>>> print(root.findtext("system/purpose"))
>>> print(root.find("systempurpose"))
```

- `iter(tag=None)`：从 xml 根节点开始，根据传入的元素的 tag 返回所有的元素集合的迭代器

```
>>> for i in tree.iter(tag = "command"):
    print(i.text)
```

- `iterfind(match)` ：根据传入的 tag 名称或者 path 以迭代器的形式返回所有的子元素

```
>>> for i in tree.iterfind("system/purpose"):
    print(i.text)
```

Element 主要的方法和使用示例

- `tag` ：字符串，用来表示元素所代表的名称

```
>>> print(root[1].tag)    # 输出 system
```

- `text` ：表示元素所对应的具体值

```
>>> print(root[1].text)    # 输出空串
```

- `attrib` ：用字典表示的元素的属性

```
>>> print(root[1].attrib)    # 输出 {"platform": "aix", "name": "aixtest"}
```

- `get(key, default=None)` ：根据元素属性字典的 key 值获取对应的值，如果找不到对应的属性，则返回 default

```
>>> print(root[1].attrib.get("platform"))    # 输出 aix
```

- `items()` ：将元素属性以（名称，值）的形式返回

```
>>> print(root[1].items())    # [("platform", "aix"), ("name", "aixtest")]
```

- `keys()` ：返回元素属性的 key 值集合

```
>>> print(root[1].keys())    # 输出 ["platform", "name"]
```

- `find(match)` ：根据传入的 tag 名称或者 path 返回第一个对应的 element 对象，或者返回 None

- `findall(match)` ：根据传入的 tag 名称或者 path 以列表的形式返回所有符合条件的元素
- `findtext(match, default=None)` ：根据传入的 tag 名称或者 path 返回第一个对应的 element 对象对应的值，即 text 属性，如果找不到则返回 default 的设置
- `list(elem)` ：根据传入的元素的名称返回其所有的子节点

```
>>> for i in list(root.findall("system/system_type")):
    print(i.text)    # 输出 virtual virtual
```

elementree 的 iterparse 工具能够避免将整个 XML 文件加载到内存，从而解决当读入文件过大内存而消耗过多的问题。iterparse 返回一个可以迭代的由元组（时间，元素）组成的流对象，支持两个参数—— `source` 和 `events` ，其中 `event` 有 4 种选择 —— `start` 、 `end` 、 `startns` 和 `endns` （默认为 end），分别与 SAX 解析的 `startElement` 、 `endElement` 、 `startElementNS` 和 `endElementNS` 一一对应。

iterparse 的使用示例：

```
>>> count = 0
>>> for event, elem in ET.iterparse("test.xml"): # 对 iterparse 的返回值进行迭代
    if event == "end":
        if elem.tag == "userid":
            count += 1
    elem.clear()
>>> print(count)
```

建议 44：理解模块 pickle 优劣

序列化的场景很常见，如：在磁盘上保存当前程序的状态数据以便重启的时候能够重新加载；多用户或者分布式系统中数据结构的网络传输时，可以将数据序列化后发送给一个可信网络对端，接收者进行反序列化后便可以重新恢复相同的对象； `session` 和 `cache` 的存储等。

序列化，简单地说就是把内存中的数据结构在不丢失其身份和类型信息的情况下转换成对象的文本或二进制表示的过程。对象序列化后的形式经过反序列化过程应该能恢复原有对象。

Python 中有很多支持序列化的模块，如 `pickle` 、 `json` 、 `marshal` 和 `shelve` 等。

pickle 估计是最通用的序列化模块了，它还有个 C 语言的实现 cPickle，相比 pickle 来说具有较好的性能，其速度大概是 pickle 的 1000 倍，因此在大多数应用程序中应该优先使用 cPickle（注：cPickle 除了不能被继承之外，它们两者的使用基本上区别不大）。pickle 中最主要的两个函数对为 `dump()` 和 `load()` ，分别用来进行对象的序列化和反序列化。

- `pickle.dump(obj, file[, protocol])` ：序列化数据到一个文件描述符（一个打开的文件、套接字等）。参数 obj 表示需要序列化的对象，包括布尔、数字、字符串、字节数

组、None、列表、元组、字典和集合等基本数据类型，此外 pickle 还能够处理循环，递归引用对象、类、函数以及类的实例等。参数 file 支持 `write()` 方法的文件句柄，可以为真实的文件，也可以是 `StringIO` 对象等。protocol 为序列化使用的协议版本，0 表示 ASCII 协议，所序列化的对象使用可打印的 ASCII 码表示；1 表示老式的二进制协议；2 表示 2.3 版本引入的新二进制协议，比以前的更高效。其中协议 0 和 1 兼容老版本的 Python。protocol 默认值为 0。

- `load(file)`：表示把文件中的对象恢复为原来的对象，这个过程也被称为反序列化。

```
>>> import cPickle as pickle
>>> my_data = {"name": "Python", "type": "Language", "version": "2.7.5"}
>>> fp = open("picklefile.dat", "wb")    # 打开要写入的文件
>>> pickle.dump(my_data, fp)    # 使用 dump 进行序列化
>>> fp.close()
>>>
>>> fp = open("picklefile.dat", "rb")
>>> out = pickle.load(fp)    # 反序列化
>>> print(out)
>>> fp.close()
```

pickle 之所以能成为通用的序列化模块，与其良好的特性是分不开的，总结为以下几点：

- 接口简单，容易使用。使用 `dump()` 和 `load()` 便可轻易实现序列化和反序列化。
- pickle 的存储格式具有通用性，能够被不同平台的 Python 解析器共享，比如 Linux 下序列化的格式文件可以在 Windows 平台的 Python 解析器上进行反序列化，兼容性较好。
- 支持的数据类型广泛。如数字、布尔值、字符串，只包含可序列化对象的元组、字典、列表等，非嵌套的函数、类以及通过类的 `__dict__` 或者 `__getstate__()` 可以返回序列化对象的实例等。
- pickle 模块是可以扩展的。对于实例对象，pickle 在还原对象的时候一般是不调用 `__init__()` 函数的，如果要调用 `__init__()` 进行初始化，对于古典类可以在类定义中提供 `__getinitargs__()` 函数，并返回一个元组，当进行 unpickle 的时候，Python 就会自动调用 `__init__()`，并把 `__getinitargs__()` 中返回的元组作为参数传递给 `__init__()`，而对于新式类，可以提供 `__getnewargs__()` 来提供对象生成时候的参数，在 unpickle 的时候以 `Class.__new__(Class, *arg)` 的方式创建对象。对于不可序列化的对象，如 sockets、文件句柄、数据库连接等，也可以通过实现 pickle 协议来解决这些巨献，主要是通过特殊方法 `__getstate__()` 和 `__setstate__()` 来返回实例在被 pickle 时的状态。

示例：

```
import cPickle as pickle
class TextReader:
    def __init__(self, filename):
        self.filename = filename    # 文件名称
        self.file = open(filename)    # 打开文件的句柄
        self.postion = self.file.tell()    # 文件的位置

    def readline(self):
        line = self.file.readline()
        self.postion = self.file.tell()
        if not line:
            return None
        if line.endswith("\n"):
            line = line[:-1]
        return "{}: {}".format(self.postion, line)

    def __getstate__(self):    # 记录文件被 pickle 时候的状态
        state = self.__dict__.copy()    # 获取被 pickle 时的字典信息
        del state["file"]
        return state

    def __setstate__(self, state):    # 设置反序列化后的状态
        self.__dict__.update(state)
        file = open(self.filename)
        self.file = file

reader = TextReader("zen.text")
print(reader.readline())
print(reader.readline())
s = pickle.dumps(reader)    # 在 dumps 的时候会默认调用 __getstate__
new_reader = pickle.loads(s)    # 在 loads 的时候会默认调用 __setstate__
print(new_reader.readline())
```

- 能够自动维护对象间的引用，如果一个对象上存在多个引用，pickle 后不会改变对象间的引用，并且能够自动处理循环和递归引用。

```
>>> a = ["a", "b"]
>>> b = a    # b 引用对象 a
>>> b.append("c")
>>> p = pickle.dumps((a, b))
>>> a1, b1 = pickle.loads(p)
>>> a1
["a", "b", "c"]
>>> b1
["a", "b", "c"]
>>> a1.append("d")    # 反序列化对 a1 对象的修改仍然会影响到 b1
>>> b1
["a", "b", "c", "d"]
```

但 pickle 使用也存在以下一些限制：

- pickle 不能保证操作的原子性。pickle 并不是原子操作，也就是说在一个 pickle 调用中如果发生异常，可能部分数据已经被保存，另外如果对象处于深递归状态，那么可能超出 Python 的最大递归深度。递归深度可以通过 `sys.setrecursionlimit()` 进行扩展。
- pickle 存在安全性问题。Python 的文档清晰地表明它不提供安全性保证，因此对于一个从不可信的数据源接收到的数据不要轻易进行反序列化。由于 `loads()` 可以接收字符串作为参数，精心设计的字符串给入侵提供了一种可能。在 Python 解释器中输入代码 `pickle.loads("cos\nsystem\n(S'dir\ntR.")` 便可以查看当前目录下所有文件。可以将 `dir` 替换为其他更具破坏性的命令。如果要进一步提高安全性，用户可以通过继承类 `pickle.Unpickler` 并重写 `find_class()` 方法来实现。
- pickle 协议是 Python 特定的，不同语言之间的兼容性难以保障。用 Python 创建的 pickle 文件可能其他语言不能使用。

建议 45：序列化的另一个不错的选择——JSON

JSON（JavaScript Object Notation）是一种轻量级数据交换格式，它基于 JavaScript 编程语言的一个子集，于 1999 年 12 月成为一个完全独立于语言的文本格式。其格式使用了许多其他流行编程的约定，简单灵活，可读性和互操作性较强、易于解析和使用。

Python 中有一系列的模块提供对 JSON 格式的支持，如 `simplejson`、`cjson`、`yajl`、`ujson`，自 Python2.6 后又引入了标准库 JSON。简单来说 `cjson` 和 `ujson` 是用 C 来实现的，速度较快。据 `cjson` 的文档表述：其速率比纯 Python 实现的 json 模块大概要快 250 倍。`yajl` 是 Cpython 版本的 JSON 实现，而 `simplejson` 和标准库 JSON 本质来说无多大区别，实际上 Python2.6 中的 json 模块就是 `simplejson` 减去对 Python2.4、Python2.5 的支持以充分利用最新的兼容未来的功能。不过相对于 `simplejson`，标准库更新相对较慢。在实际应用过程中将这两者结合较好的做法是采用如下 import 方法：

```
try:
    import simplejson as json
except ImportError:
    import json
```

Python 的标准库 JSON 提供的最常用的方法与 pickle 类似，`dump/dumps` 用来序列化，`load/loads` 用来反序列化。需要注意 json 默认不支持非 ASCII-based 的编码，如 load 方法可能在处理中文字符时不能正常显示，则需要通过 encoding 参数指定对应的字符编码。在序列化方面，相比 pickle，JSON 具有以下优势：

- 使用简单，支持多种数据类型。JSON 文档的构成非常简单，仅存在以下两大数据结构：
    - 名称/值对的集合。在各种语言中，它被实现为一个对象、记录、结构、字典、散列

  表、键列表或关联数组。
  - 值的有序列表。在大多数语言中，它被实现为数组、向量、列表或序列。在 Python 中对应支持的数据类型包括字典、列表、字符串、整数、浮点数、True、False、None 等。JSON 中数据结构和 Python 中的转换并不是完全一一对应，存在一定的差异。
- 存储格式可读性更为友好，容易修改。相比于 pickle 来说，json 格式更加接近程序员的思维，阅读和修改上要容易得多。`dumps()` 函数提供了一个参数 indent 使生成的 json 文件可读性更好，0 意味着“每个值单独一行”；大于 0 的数字意味着“每个值单独一行并且使用这个数字的空格来缩进嵌套的数据结构”。但需要注意的是，这个参数是以文件大小变大为代价的。

- json 支持跨平台跨语言操作，能够轻易被其他语言解析，如 Python 中生成的 json 文件可以轻易使用 JavaScript 解析，互操作性更强，而 pickle 格式的文件只能在 Python 语言中支持。此外 json 原生的 JavaScript 支持，客户端浏览器不需要为此使用额外的解释器，特别适用于 Web 应用提供快速、紧凑、方便地序列化操作。此外，相比于 pickle，json 的存储格式更为紧凑，所占空间更小。

- 具有较强的扩展性。json 模块还提供了编码（JSONEncoder）和解码类（JSONDecoder）以便用户对其默认不支持的序列化类型进行扩展。

- json 在序列化 datetime 的时候会抛出 TypeError 异常，这是因为 json 模块本身不支持 datetime 的序列化，因此需要对 json 本身的 JSONEncoder 进行扩展。有多种方法可以实现：

```
import datetime
from time import mktime
try:
    import simplejson as json
except ImportError:
    import json

class DateTimeEncoder(json.JSONEncoder):    # 为 JSONEncoder 进行扩展
    def default(self, obj):
        if isinstance(obj, datetime.datetime):
            return obj.strftime("%Y-%m-%d %H:%M:%S")
        elif isinstance(obj, date):
            return obj.strftime("%Y-%m-%d")
        return json.JSONEncoder.default(self, obj)

d = datetime.datetime.now()
print(json.dumps(d, cls=DateTimeEncoder))    # 使用 cls 指定编码器的名称
```

Python 中标准模块 json 的性能比 pickle 与 cPickle 稍逊。如果对序列化性能要求非常高的场景，可以使用 cPickle 模块。

建议 46：使用 traceback 获取栈信息

面对异常开发人员最希望看到的往往是异常发生时候的现场信息，`traceback` 模块可以满足这个需求，它会输出完整的栈信息。

```
except IndexError as ex:
    print("Sorry, Exception occured, you accessed an element out of range")
    print(ex)
    traceback.print_exc()
```

程序会输出异常发生时候完整的栈信息，包括调用顺序、异常发生的语句、错误类型等。

`traceback.print_exc()` 方法打印出的信息包括 3 部分：错误类型（IndexError）、错误对应的值（list index out of range）以及具体的 trace 信息，包括文件名、具体的行号、函数名以及对应的源代码。`Traceback` 模块提供了一系列方法来获取和显示异常发生时候的 trace 相关信息：

- `traceback.print_exception(type, value, traceback[, limit[, file]])`，根据 limit 的设置打印栈信息，file 为 None 的情况下定位到 `sys.stderr`，否则则写入文件；其中 type、value、traceback 这 3 个参数对应的值可以从 `sys.exc_info()` 中获取。
- `traceback.print_exc([limit[, file]])`，为 `print_exception()` 函数的缩写，不需要传入 type、value、traceback 这 3 个参数。
- `traceback.format_exc([limit])`，与 `print_exec()` 类似，区别在于返回形式为字符串。
- `traceback.extract_stack([file, [limit]])`，从当前栈帧中提取 trace 信息。

可以参看 Python 文档获取更多关于 traceback 所提供的抽取、格式化或者打印程序运行时候的栈跟踪信息的方法。本质上模块 traceback 获取异常相关的数据都是通过 `sys.exc_info()` 函数得到的。当有异常发生的时候，该函数以元组的形式返回 `(type, value, traceback)`，其中 type 为异常的类型，value 为异常本身，traceback 为异常发生时候的调用和堆栈信息，它是一个 traceback 对象，对象中包含出错的行数、位置等数据。

```
tb_type, tb_val, exc_tb = sys.exc_info()
for filename, linenum, funcname, source in traceback.extract_tb(exc_tb):
    print("%-23s:%s '%s' in %s()" % (filename, linenum, source, funcname))
```

实际上除了 traceback 模块本身，inspect 模块也提供了获取 traceback 对象的接口，`inspect.trace([context])` 可以返回当前帧对象以及异常发生时进行捕获的帧对象之间的所有栈帧记录，因此第一个记录代表当前调用对象，最后一个代表异常发生时候的对象。其中每一个列表元素都是一个由 6 个元素组成的元组：（frame 对象，文件名，当前行号，函数名，源代码列表，当前行在源代码列表中的位置）。

此外如果想进一步追踪函数调用的情况，还可以通过 inspect 模块的 `inspect.stack()` 函数查看函数层级调用的栈相关信息。因此，当异常发生的时候，合理使用上述模块中的方法可以快速地定位程序中的问题所在。

建议 47：使用 logging 记录日志信息

仅仅将信息输出到控制台是远远不够的，更为常见的是使用日志保存程序运行过程中的相关信息，如运行时间、描述信息以及错误或者异常发生时候的特定上下文信息。Python 中自带的 logging 模块提供了日志功能，它将 logger 的 level 分为 5 个级别，可以通过 `Logger.setLevel(lvl)` 来设置，其中 DEBUG 为最低级别，CRITICAL 为最高级别，默认的级别为 WARNING。

Level	使用情形
DEBUG	详细的信息，在追踪问题的时候使用
INFO	正常的信息
WARNING	一些不可预见的问题发生，或者将要发生，如磁盘空间低等，但不影响程序的运行
ERROR	由于某些严重的问题，程序中的一些功能受到影响
CRITICAL	严重的错误，或者程序本身不能够继续运行

logging lib 包含以下 4 个主要对象：

- logger：logger 是程序信息输出的接口，它分散在不同的代码中，使得程序可以在运行的时候记录相应的信息，并根据设置的日志级别或 filter 来决定哪些信息需要输出，并将这些信息分发到其关联的 handler。常用的方法有 `Logger.setLevel()`、`Logger.addHandler()`、`Logger.removeHandler()`、`Logger.addFilter()`、`Logger.debug()`、`Logger.info()`、`Logger.warning()`、`Logger.error()`、`etLogger()` 等。
- Handler：Handler 用来处理信息的输出，可以将信息输出到控制台、文件或者网络。可以通过 `Logger.addHandler()` 来给 logger 对象添加 handler，常用的 handler 有 StreamHandler 和 FileHandler 类。StreamHandler 发送错误信息到流，而 FileHandler 类用于向文件输出日志信息，这两个 handler 定义在 logging 的核心模块中。其他的 handler 定义在 `logging.handles` 模块中，如 `HTTPHhandler`、`SocketHandler`。
- Formatter：决定 log 信息的格式，格式使用类似于 `%(< dictionary key >)s` 的形式来定义，如 `'%(asctime)s - %(levelname)s - %(message)s'`，支持的 key 可以在 Python 自带的文档 LogRecord attributes 中查看
- Filter：用来决定哪些信息需要输出。可以被 handler 和 logger 使用，支持层次关系，比如，如果设置了 filter 名称为 A.B 的 logger，则该 logger 和其子 logger 的信息会被输出，如 A.B、A.B.C

`logging.basicConfig([**kwargs])` 提供对日志系统的基本配置，默认使用 StreamHandler 和 Formatter 并添加到 root logger，该方法自 Python2.4 开始可以接受字典参数，支持的字典参数：

格式	描述
filename	指定 FileHandler 的文件名，而不是默认的 StreamHandler
filemode	打开文件的模式，同 open 函数中的同名参数，默认为 'a'
format	输出格式字符串
datefmt	日期格式
level	设置根 logger 的日志级别
stream	指定 StreamHandler。这个参数若与 filename 冲突，忽略 stream

结合 traceback 和 logging，记录程序运行过程中的异常：

```
import traceback
import sys
import logging
gList = ["a", "b", "c", "d", "e", "f", "g"]
logging.basicConfig( # 配置日志的输出方式及格式
    level = logging.DEBUG,
    filename = "log.txt",
    filemode = "w",
    format = "%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s % (message)s",
)

def f():
    gList[5]
    logging.info("[INFO]:calling method g() in f()")    # 记录正常的信息
    return g()

def g():
    logging.info("[INFO]:calling method h() in g()")
    return h()

def h():
    logging.info("[INFO]:Delete element in gList in h()")
    del gList[2]
    logging.info("[INFO]:calling method i() in h()")
    return i()

def i():
    logging.info("[INFO]:Append element i to gList in i()")
    gList.append("i")
    print(gList[7])

if __name__ == "__main__":
    logging.debug("Information during calling f():")
    try:
        f()
    except IndexError as ex:
        print("Sorry, Exception occured, you accessed an element out of range")
        # traceback.print_exc()
        ty, tv, tb = sys.exc_info()
        logging.error("[ERROR]: Sorry, Exception occured, you accessed an element out
of range")    # 记录异常错误消息
        logging.critical("object info:%s" % ex)
        logging.critical("Error Type:{0}, Error Information:{1}".format(ty, tv))    #
记录异常的类型和对应的值
        logging.critical("".join(traceback.format_tb(tb)))    # 记录具体的 trace 信息
        sys.exit(1)
```

修改程序后在控制台上对用户仅显示错误提示信息，而开发人员如果需要 debug 可以在日志文件中找到具体运行过程中的信息。

上面的代码中控制运行输出到 console 上用的是 `print()`，但这种方法比较原始，logging 模块提供了能够同时控制输出到 console 和文件的方法：

```
console = logging.StreamHandler()
console.setLevel(logging.ERROR)
formatter = logging.Formatter("%(name)-12s: %(levelname)-8s %(message)s")
console.setFormatter(formatter)
logging.getLogger('').addHandler(console)
```

为了使 Logging 使用更为简单可控，logging 支持 logging.config 进行配置，支持 dictConfig 和 fileConfig 两种形式，其中 fileConfig 是基于 `configparser()` 函数进行解析，必须包含的内容为 `[loggers]`、`[handlers]` 和 `[formatters]`。

```
[loggers]
keys=root
[logger_root]
level=DEBUG
handlers=hand01
[handlers]
keys=hand01

[handler_hand01]
class=StreamHandler
level=INFO
formatter=form01
args=(sys.stderr,)
[formatters]
keys=form01
[formatter_form01]
format=%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s
datefmt=%a, %d %b %Y %H:%M:%S
```

关于 logging 的使用，还有几点建议：

- 尽量为 logging 取一个名字而不是采用默认，这样当在不同的模块中使用的时候，其他模块只需要使用以下代码就可以方便地使用同一个 logger，因为它本质上符合单例模式：

```
import logging
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)
```

- 为了方便地找出问题所在，logging 的名字建议以模块或者 class 来命名。Logging 名称遵循按 "." 划分的继承规则，根是 root logger，logger a.b 的父 logger 对象为 a。
- Logging 只是线程安全的，不支持多进程写入同一个日志文件，因此对于多个进程，需要配置不同的日志文件。

建议 48：使用 threading 模块编写多线程程序

GIL 的存在使得 Python 多线程编程暂时无法充分利用多处理器的优势，这种限制并不意味着我们需要放弃多线程。的确，对于只含纯 Python 的代码也许使用多线程并不能提高运行速率，但在以下几种情况，如等待外部资源返回，或者为了提高用户体验而建立反应灵活的用户界面，或者多用户应用程序中，多线程仍然是一个比较好的解决方案。Python 为多线程编程提供了两个非常简单明了的模块：thread 和 threading。

thread 模块提供了多线程底层支持模块，以低级原始的方式来处理和控制线程，使用起来较为复杂；而 threading 模块基于 thread 进行包装，将线程的操作对象化，在语言层面提供了丰富的特性。Python 多线程支持用两种方式来创建线程：一种通过继承 Thread 类，重写它的 run() 方法（注意不是 start() 方法）；另一种是创建一个 thread.Thread 对象，在它的初始化函数（`__init__()`）中将可调用对象作为参数传入。实际应用中，推荐优先使用 threading 模块而不是 thread 模块。

- threading 模块对同步原语的支持更为完善和丰富。就线程的同步和互斥来说，thread 模块只提供了一种锁类型 `thread.LockType`，而 threading 模块中不仅有 Lock 指令锁，RLock 可重入指令锁，还支持条件变量 Condition、信号量 Semaphore、BoundedSemaphore 以及 Event 事件等

- threading 模块在主线程和子线程交互上更为友好，threading 中的 join() 方法能够阻塞当前上下文环境的线程，直到调用此方法的线程终止或到达指定的 timeout（可选参数）。利用该方法可以方便地控制主线程和子线程以及子线程之间的执行。

- thread 模块不支持守护线程。thread 模块中主线程退出的时候，所有的子线程不论是否还在工作，都会被强制结束，并且没有任何警告，也没有任何退出前的清理工作。 测试代码：

```
from thread import start_new_thread
import time
def myfunc(a, delay):
    print("I wll calculate square of {} after delay for {}".format(a, delay))
    time.sleep(delay)
    print("calculate begins...")
    result = a * a
    print(result)
    return result
start_new_thread(myfunc, (2, 5))     # 同时启动两个线程
start_new_thread(myfunc, (6, 8))
time.sleep(1)
```

实际上很多情况下我们可能希望主线程能够等待所有子线程都完成时才退出，这时应该使用 threading 模块，它支持守护线程，可以通过 `setDaemon()` 函数来设定线程的 daemon 属性。当 daemon 属性设置为 True 的时候表明主线程的退出可以不用等待子线

程完成。默认情况下，daemon 标志为 False，所有的非守护线程结束后主线程才会结束。

```
import threading
import time
def myfunc(a, delay):
    print("I will calculate square of {} after delay for {}".format(a, delay))
    time.sleep(delay)
    print("calculate begins...")
    result = a * a
    print(result)
    return result

t1 = threading.Thread(target=myfunc, args=(2, 5))
t2 = threading.Thread(target=myfunc, args=(6, 8))
print(t1.isDaemon())
print(t2.isDaemon())
t2.setDaemon(True)
t1.start()
t2.start()
```

- Python3 中已经不存在 thread 模块。thread 模块在 Python3 中被命名为 _thread，这种更改主要是为了进一步明确表示与 thread 模块相关的更多的是具体的实现细节，它更多展示的是操作系统层面的原始操作和处理。

建议 49：使用 Queue 使多线程编程更安全

多线程编程不是件容易的事情。线程间的同步和互斥，线程间数据的共享等这些都是涉及线程安全要考虑的问题。纵然 Python 中提供了众多的同步和互斥机制，如 mutex、condition、event 等，但同步和互斥本身就不是一个容易的话题，稍有不慎就会陷入死锁状态或者威胁线程安全。

Python 中的 Queue 模块提供了 3 种队列：

- `Queue.Queue(maxsize)` ：先进先出，maxsize 为队列大小，其值为非正数的时候为无限循环队列
- `Queue.LifoQueue(maxsize)` ：后进先出，相当于栈
- `Queue.PriorityQueue(maxsize)` ：优先级队列

这 3 种队列支持以下方法：

- `Queue.qsize()` ：返回近似的队列大小。之所以说是近似，当该值 > 0 的时候并不保证并发执行的时候 get() 方法不被阻塞，同样，对于 put() 方法有效。
- `Queue.empty()` ：队列为空的时候返回 True，否则返回 False
- `Queue.full()` ：当设定了队列大小的情况下，如果队列满则返回 True，否则返回 False。

- `Queue.put(item[, block[, timeout]])`：往队列中添加元素 item，block 设置为 False 的时候，如果队列满则抛出 Full 异常。如果 block 设置为 True，timeout 为 None 的时候则会一直等待直到有空位置，否则会根据 timeout 的设定超时后抛出 Full 异常。
- `Queue.put_nowait(item)`：等于 `put(item, False).block` 设置为 False 的时候，如果队列空则抛出 Empty 异常。如果 block 设置为 True、timeout 为 None 的时候则会一直等到有元素可用，否则会根据 timeout 的设定超时后抛出 Empty 异常。【个人：这里是不是说反了？】
- `Queue.get([block[, timeout]])`：从队列中删除元素并返回该元素的值
- `Queue.get_nowait()`：等价于 `get(False)`
- `Queue.task_done()`：发送信号表明入列任务已经完成，经常在消费者线程中用到
- `Queue.join()`：阻塞直至队列中所有的元素处理完毕

Queue 模块实现了多个生产者多个消费者的队列，当多线程之间需要信息安全的交换的时候特别有用，因此这个模块实现了所需要的锁原语，为 Python 多线程编程提供了有力的支持，它是线程安全的。需要注意的是 Queue 模块中的队列和 `collections.deque` 所表示的队列并不一样，前者主要用于不同线程之间的通信，它内部实现了线程的锁机制；而后者主要是数据结构上的概念，因此支持 in 方法。

因为 queue 本身能够保证线程安全，因此不需要额外的同步机制。下面有个多线程下载的例子：

```
import os
import Queue
import threading
import urllib2
class DownloadThread(threading.Thread):
    def __init__(self, queue):
        threading.Thread.__init__(self)
        self.queue = queue
    def run(self):
        while True:
            url = self.queue.get()    # 从队列中取出一个 url 元素
            print(self.name + "begin download" + url + "...")
            self.download_file(url)    # 进行文件下载
            self.queue.task_done()    # 下载完毕发送信号
            print(self.name + " download completed!!!")
    def download_file(self, url):    # 下载文件
        urlhandler = urllib2.urlopen(url)
        fname = os.path.basename(url) + ".html"    # 文件名称
        with open(fname, "wb") as f:    # 打开文件
            while True:
                chunk = urlhandler.read(1024)
                if not chunk:
                    break
                f.write(chunk)
if __name__ == "__main__":
    urls = ["https://www.createspace.com/3611970","http://wiki.python.org/moni.WebProg
ramming"]
    queue = Queue.Queue()
    # create a thread pool and give them a queue
    for i in range(5):
        t = DownloadThread(queue)    # 启动 5 个线程同时进行下载
        t.setDaemon(True)
        t.start()

    # give the queue some data
    for url in urls:
        queue.put(url)

    # wait for the queue to finish
    queue.join()
```

第 5 章 设计模式

软件开发行业的设计模式广为人知，这是 GoF 的《设计模式——可复用面向对象软件的基础》的功劳，后来的《Head First 设计模式》则通过幽默的文风使其广泛流行于程序员之间。但是这两本分别使用 C++ 和 Java 编程语言作为载体，不能直接照搬到 Python 程序中，否则会有静态语言风格。

Python 的动态语言特性并不能完全替代设计模式。

建议 50：利用模块实现单例模式

在 GoF 的 23 种设计模式中，单例是最常使用的模式，通过单例模式可以保证系统中一个类只有一个实例而且该实例易于被外界访问，从而方便对实例个数的控制并节约系统资源。每当大家想要实现一个 XxxManager 的类时，往往意味着这是一个单例。

有不少现代编程语言将其加到了语言特性中，如 scala 和 falcon 语言都把 object 定义成关键词，并用其声明单例。如在 scala 中，一个单例如下：

```
object Singleton {
  def show = println("I am a singleton")
}
```

object 定义了一个名为 Singleton 的单例，它满足单例的 3 个需求：一是只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。对于第三点，在任何地方都可以通过调用 `Singleton.show()` 来验证。在 scala 中，单例没有显式的初始化操作，但并不是所有在语法层面支持单例模式的编程语言都如此，比如 falcon 就不一样。

```
object object_name [from class1, class2 ... classN]
    property_1 = expression
    property_2 = expression
    ...
    property_N = expression
    [init block]
    function method_1([parameter_list])
        [method_body]
    end
    ...
    function method_N([parameter_list])
        [method_body]
    end
end
```

`[init block]` 能够让程序员手动控制单例的初始化代码。但是与 scala 和 falcon 相比，动态语言 Python 缺乏声明私有构造函数的语法元素，实例又带有类型信息。所以以下方法是不可行的：

```
class _Singleton(object):
    pass
Singleton = _Singleton()
del _Singleton    # 试图删除 class 定义
another = Singleton.__class__()    # 没用，绕过！
print(type(another))
# 输出
<class '__main__._Singleton'>
```

可见虽然把 Singleton 的类定义删除了，但仍然有办法通过已有实例的 `__class__` 属性生成一个新的实例。于是许多 Pythonista 把目光聚集到真正创建实例的方法 `__new__` 上：

```
class Singleton(object):
    _instance = None
    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__(cls, *args, **kwargs)
        return cls._instance
if __name__ = '__main__':
    s1 = Singleton()
    s2 = Singleton()
    assert id(s1) == id(s2)
```

这个方法基本上可以保证“只能有一个实例”的要求了，但是在并发情况下可能会发生意外，解决办法是引入锁：

```
class Singleton(object):
    objs = {}
    objs_locker = threading.Lock()
    def __new__(cls, *args, **kwargs):
        if cls in cls.objs:
            return cls.objs[cls]
        cls.objs_locker.acquire()
        try:
            if cls in cls.objs:    ## double check locking
                return cls.objs[cls]
            cls.objs[cls] = object.__new__(cls)
        finally:
            cls.objs_locker.release()
```

利用经典的双检查锁机制，确保了在并发环境下 Singleton 的正确实现。但这个方案并不完美，比如以下两个问题：

- 如果 Singleton 的子类重载了 `__new__()` 方法，会覆盖或者干扰 Singleton 类中 `__new__()` 的执行，虽然这种情况出现的概率极小，但不容忽视。
- 如果子类有 `__init__()` 方法，那么每次实例化该 Singleton 的时候，`__init__()` 都会被调用到，这显然是不应该的，`__init__()` 只应该在创建实例的时候被调用一次。

这两个问题当然可以解决，比如通过文档告知其他程序员，子类化 Singleton 的时候，务必调用父类的 `__new__()` 方法；而第二个问题也可以通过偷偷地替换掉 `__init__()` 方法来确保它只调用一次。但是，为了实现一个单例，做大量的、水面之下的工作相当不 Pythonic。

模块采用的其实是天然的单例的实现方式：

- 所有的变量都会绑定到模块
- 模块只初始化一次
- import 机制是线程安全的（保证了在并发状态下模块也只有一个实例）

所以创建一个 world 单例时：

```
# World.py
import Sun
def run():
    while True:
        Sun.rise()
        Sun.set()
```

然后在入口文件 `main.py` 里导入，并调用 `run()` 函数：

```
# main.py
import World
World.run()
```

> ### 注意
>
> Alex Martelli 认为单例模式要求“实例的唯一性”本身是有问题的，实际更值得关注的是实例的状态，只要所有的实例共享状态（可以狭义地理解为属性）、行为（可以狭义地理解为方法）一致就可以了。于是有 Borg 模式（在 C# 中又称为 Monostate 模式）。
>
> ```
> class Borg:
>     __shared_state = {}
>     def __init__(self):
>         self.__dict__ = self.__shared_state
>     # and whatever else you want in your class -- that's all!
> ```
>
> 通过 Borg 模式，可以创建任意数量的实例，但因为它们共享状态，从而保证了行为一致。Alex 的这个 Borg 模式仅适用于古典类（classic classess），Python2.2 以后的新式类（new-style classes）需要使用 `__getattr__` 和 `__setattr__` 方法来实现。

建议 51：用 mixin 模式让程序更加灵活

先来了解一下模版方法模式，模版方法模式就是在一个方法中定义一个算法的骨架，并将一些实现步骤延迟到子类中。模版方法可以使子类在不改变算法结构的情况下，重新定义算法中的某些步骤。

模版方法在 C++ 或其他语言中并无不妥，但在 Python 中有点画蛇添足。比如模版方法，需要先定义一个基类，而实现行为的某些步骤则必须在其子类中，在 Python 中并无必要。

```
class People(object):
    def make_tea(self):
        teapot = self.get_teapot()
        teapot.put_in_tea()
        teapot.put_in_water()
        return teapot
```

`get_teapot()` 方法并不需要预先定义：

```
class OfficePeople(People):
    def get_teapot(self):
        return SimpleTeaPot()

class HomePeople(People):
    def get_teapot(self):
        return KungfuTeapot()
```

虽然看起来像模板方法，但是基类并不需要预先声明抽象方法，甚至还带来吊事代码的便利。

如果子类没有实现 `get_teapot()` 方法，所以一调用 `make_tea()` 就会产生一个找不到方法的 `AttributeError` 。

但是，这样导致方法只能实现一个，解决方法有两种：一种是继承子类，再重写 `get_teapot()` ；另一个则是把 `get_teapot()` 方法提取出来，把它以多继承的方式做一次静态混入。

```
class UseSimpleTeapot(object):
    def get_teapot(self):
        return SimpleTeaPot()

class UseKungfuTeapot(object):
    def get_teapot(self):
        return KungfuTeapot()

class OfficePeople(People, UseSimpleTeapot):
    pass
class HomePeople(People, UseKungfuTeapot):
    pass
class Boss(People, UseKungfuTeapot):
    pass
```

但是这样的代码仍然没有把 Python 的动态性表现出来，当新的需求出现时，需要更改类定义。

于是我们开始寄望于动态地生成不同的实例：

```
def simple_tea_people():
    people = People()
    people.__bases__ += (UseSimpleTeapot, )
    return people
def coffee_people():
    people = People()
    people.__bases__ += (UseCoffeepot, )
    return people
def tea_and_coffee_people():
    people = People()
    people.__bases__ += (UseSimpleTeapot, UseCoffeepot, )
    return people
def boss():
    people = People()
    people.__bases__ += (KungfuTeapot, UseCoffeepot, )
    return people
```

这个代码能够运行的原理是，每个类都有一个 `__bases__` 属性，它是一个元组，用来存放所有的基类。与其他静态语言不同，Python 语言中的基类在运行中可以动态改变。所以当我们向其中添加新的基类时，这个类就拥有了新的方法，也就是所谓的混入（mixin）。这种动态性的好处在于代码获得了更丰富的扩展功能。

值得进一步探索的是，利用反射技术，甚至不需要修改代码：

```
import mixins
def staff():
    people = People()
    bases = []
    for i in config.checked():
        bases.append(getattr(mixins, i))
    people.__bases__ += tuple(bases)
    return people
```

通过这个框架代码，OA 系统的开发人员只需要把常见的需求定义成 Mixin 预告放在 mixins 模块中，就可以在不修改代码的情况下通过管理界面满足几乎所有需求了。

建议 52：用发布订阅模式实现松耦合

发布订阅模式（publish/subscribe 或 pub/sub）是一种编程模式，消息的发送者（发布者）不会发送器消息给特定的接收者（订阅者），而是将发布的消息分为不同的类别直接发布，并不会关注订阅者是谁。而订阅者可以对一个或多个类别感兴趣，且只接收感兴趣的消息，并且不关注是哪个发布者发布的消息。这种发布者和订阅者的解耦可以允许更好地可扩放性和更为动态的网络拓扑。

发布订阅模式的优点是发布者与订阅者松散的耦合，双方不需要知道对方的存在。由于主题是被关注的，发布者和订阅者可以对系统拓扑毫无所知。无论对方是否存在，发送者和订阅者都可以继续正常操作。要实现这个模式，就需要有一个中间代理人，在实现中一般被称为 Broker，它维护着发布者和订阅者的关系：定于这把感兴趣的主题告诉它，而发布者的信息也通过它路由到各个订阅者处。

简单的实现如下：

```
from collections import defaultdict
route_table = defaultdict(list)
def sub(self, topic, callback):
    if callback in route_table[topic]:
        return
    route_table[topic].append(callback)
def pub(self, topic, *a, **kw):
    for func in route_table[topic]:
        func(*a, **kw)
```

直接放在一个叫 `Broker.py` 的模块中（单件），省去了各种参数检测、优先处理的需求等，甚至没有取消订阅的函数，但它展现了发布订阅模式实现的最基础的结构。它的应用代码：

```
import Broker
def greeting(name):
    print("Hello, {}".format(name))
Broker.sub("greet", greeting)
Broker.pub("greet", "LaiYonghao")
```

相对于这个简化版本，blinker 和 python-message 两个模块的实现要完备得多。blinker 已经被用在了多个广受欢迎的项目上，比如 flask 和 django；而 python-message 则支持更多丰富的特性。

安装 python-message： `pip install message`

验证：

```
import message
def hello(name):
    print("Hello {}".format(name))
message.sub("greet", hello)
message.pub("greet", "lai")
```

假设现在有两个模块使用不同的形式进行日志输出，于是可以这么编写 `bar()` 函数：

```
import message
LOG_MSG = ("log", "foo")
def bar():
    message.pub(LOG_MSG, "Haha, Calling bar().")
    do_sth()
```

在已有的项目中，只需要在项目开始处加上这样的代码，继续把日志放到标准输出：

```
import message
import foo
def handle_foo_log_msg(txt):
    print(txt)
message.sub(foo.LOG_MSG, handle_foo_log_msg)
```

而在那个使用 logging 的新项目中，则这样修改：

```
def handle_foo_log_msg(txt):
    import logging
    logging.debug(txt)
```

甚至在一些不关注底层库的日志项目中，直接无视就可以了。通过 message，可以轻松获得库与应用之间的解耦，因为库关注的是要有日志，而不关注日志输出到哪里；应用关注的是日志要统一放置，但不关系谁往日志文件中输出内容，这与发布订阅模式类似。

除了简单的 `sub()/pub()` 之外，python-message 还支持取消订阅（`unsub()`）和中止消息传递。

```
import message
def hello(name):
    print("hello {}".format(name))
    ctx = message.Context()
    ctx.discontinued = True
    return ctx
def hi(name):
    print("u cann't c me.")
message.sub("greet", hello)
message.sub("greet", hi)
message.pub("greet", "lai")
```

python-message 利用回调函数的返回值来实现消息传递。这里消息在调用 `hello()` 后就中止传递了（Broker 使用 list 对象存储回调函数就是为了保证次序）

python-message 是同步调用回调函数的，也就是说谁先 sub 谁就先被调用。大部分情况下这样已经能够满足大部分需求，但有时需要后 sub 的函数先被调用，这时 `message.sub` 函数通过一个默认参数来支持，只需要在调用 `sub` 的时候加上 `front=True`，这个回调函数将被插到所有之前已经 sub 的回调函数之前：`sub("greet", hello, front=True)`。

订阅/发布模式是观察者模式的超集，它不关注消息是谁发布的，也不关心消息由谁处理。如果需要自己的类也能够方便地订阅/发布消息，也就是想退化为观察者模式，python-message 同样提供了支持：

```
from message import observable
def greet(people):
    print("hello, {}".format(people.name))
@observable
class Foo(object):
    def __init__(self, name):
        print("Foo")
        self.name = name
        self.sub("greet", greet)
    def pub_greet(self):
        self.pub("greet", self)
foo = Foo("lai")
foo.pub_greet()
```

python-message 提供了类装饰函数 `observable()`，任何 class 只需要通过它装饰一下就拥有了 `sub/ubsub/pub/declare/retract` 等方法，它们的使用方法跟全局函数是类似的。

> ### 注意
>
> 因为 python-message 的消息订阅默认是全局性的，所以有可能产生名字冲突。在减少名字冲突方面，可以借鉴 `java/actionscript3` 的 package 起名策略，比如在应用中定义消息主题常量 `FOO='com.googlecode.python-message.FOO'`，这样多个库同时定义 FOO 常量也不容易冲突。除此之外，就是使用 uuid：
>
> ```
> uuid = 'bd1321321312321312321'
> FOO = uuid + "FOO"
> ```

建议 53：用状态模式美化代码

状态模式，就是当一个对象的内在状态改变时允许改变其行为，但这个对象看起来像是改变了其类。状态模式主要用于控制一个对象状态的条件表达式过于复杂的情况，其可把状态的判断逻辑转移到表示不同状态的一系列类中，进而把复杂的判断逻辑简化。

由于 Python 语言的动态性，状态模式的 Python 实现与 C++ 等语言的版本比起来简单得多。

```
def workday():
    print("work hard!")
def weekend():
    print("play harder!")
class People(object):
    pass
people = People()
while True:
    for i in xrange(1, 8)
        if i == 6:
            people.day = weekend
        if i == 1:
            people.day = workday
        people.day()
```

通过在不同的条件下将实例的方法（即行为）替换掉，就实现了状态模式。但仍然有缺陷：

- 查询对象的当前状态很麻烦
- 状态切换时需要对原状态做一些清扫工作，而对新的状态需要做一些初始化工作，因为每个状态需要做的事情不同，全部写在切换状态的代码中必然重复，所以需要一个机制来简化。

python-state 包通过几个辅助函数和修饰函数很好地解决了这个问题，并且定义了一个简明状态机框架：

安装：`pip install state`

然后改写：

```
from state import curr, switch, stateful, State, behavior
@stateful
class People(object):
    class Workday(State):
        default = True
        @behavior
        def day(self):
            print("work hard!")
    class Weekend(State):
        @behavior
        def day(self):
            print("play harder!")
people = People()
while True:
    for i in xrange(1, 8):
        if i == 6:
            switch(people, People.Weekend)
        if i == 1:
            switch(people, People.Workday)
        people.day()
```

首先是 `@stateful` 这个修饰函数，其中最重要的是重载了被修饰类的 `__getattr__()` 方法从而使得 People 的实例能够调用当前状态类的方法。被 `@stateful` 修饰后的类的实例是带有状态的，能有使用 `curr()` 查询当前状态，也可以使用 `switch()` 进行状态切换。

可以看到类 Workday 继承自 State 类，这个 State 类也是来自于 state 包，从其派生的子类能够使用 `__begin__` 和 `__end__` 状态转换协议，通过重载这两个协议，子类能够自定义进入和离开当前状态时对宿主的初始化和清理工作。对于一个 `@stateful` 类而言，有一个默认的状态（即其实例初始化后的第一个状态），通过类定义的 default 属性标识，defalut 设置为 True 的类成为默认状态。`@behavior` 修饰函数用以修饰状态类的方法，其实它是内置函数 `staticmethod` 的别名。

之所以将状态类的方法实现为静态方法，这是因为 state 包的原则是状态类只有行为，没有状态（状态都保存在宿主上），这样可以更好地实现代码重用。然而既然 `day()` 方法是静态的，却有 self 参数。这其实使因为 self 并不是 Python 的关键字，在这里使用 self 有助于理解状态类的宿主是 People 的实例。

通过状态模式，可以像 decorator 一样去掉 `if...raise...` 上下文判断，而且真的是一个 `if...raise...` 都没有了。另外，需要多重判断的时候要给一个方法戴上多个装饰函数的情况也没有了，还通过把多个方法分派到不同的状态类，消灭掉巨类，保持类的短小，更容易维护和重用。而且还有一个好处：当调用当前状态不存在的行为时，出错信息抛出的是 AttributeError，从而避免把问题变为复杂的逻辑错误，让程序员更容易找到出错位置，进而修正问题。

```
@stateful
class User(object):
    class NeedSignin(State):
        default = True
        @behavior
        def signin(self, usr, pwd):
            ...
            switch(self, Player.Signin)
    class Signin(State):
        @behavior
        def move(self, dst):
            ...
        @behavior
        def atk(self, other):
            ....
```

第 6 章 内部机制

建议 54：理解 built-in objects

Python 中一切皆对象：字符是对象，列表是对象，内建类型（ `built-in type` ）也是对象；用户定义的类型是对象，object 是对象，type 也是对象。自 Python2.2 以后，为了弥补内建类型和古典类（classic classes）之间的鸿沟引入了新式类（new-style classes）。在新式类中，object 是所有内建类型的基类，用户所定义的类可以继承自 object 也可以继承自内建类型。

> ### 鸿沟
>
> 在 2.2 版本之前，类和类型并不统一，如 a 是古典类 ClassA 的一个实例，那么 `a.__class__` 返回 `class__main__ClassA` ，type(a) 返回 `<type 'instance'>` 。当引入新类后，比如 ClassB 是个新类，b 是 ClassB 的实例， `b.__class__` 和 `type(b)` 都是返回 `class__main__.ClassB` 。

一些结论：

- 在 Python 中一切皆对象，type 也是对象
- `object` 和古典类没有基类， `type` 的基类为 `object`
- 新式类中 `type()` 的值和 `__class__` 的值是一样的，但古典类中实例的 `type` 为 `instance` ，其 `type()` 的值和 `__class__` 的值不一样
- 继承自内建类型的用户类的实例也是 `object` 的实例， `object` 是 `type` 的实例， `type` 实际是个元类（metaclass）
- `object` 和内建类型以及所有基于 `type` 构建的用户类都是 `type` 的实例
- 在古典类中，所有用户定义的类的类型都为 `instance`

古典类和新式类的一个区别是：新式类继承自 `object` 类或者内建类型。我们不能简单地从定义的形式上来判断一个类是新式类还是古典类（ `__metaclass__` 属性会影响到），应当通过元类的类型来确定类的类型：古典类的元类为 `types.ClassType` ，新式类的元类为 `type` 类。

新式类相对于古典类来说有很多优势：能够基于内建类型构建新的用户类型，支持 property 和描述符特性等。

作为新式类的祖先，Object 类中还定义了一些特殊方法，如： `__new__()、__init__()、__delattr__()、__getattribute__()、__setattr__()、__hash__()、__repr__()、__str__()` 等。object 的子类可以对这些方法进行覆盖以满足自身的特殊需求。

建议 55：`__init__()` 不是构造方法

从表面上看它确实很像构造方法：当需要实例化一个对象的时候，使用 `a=Class(args...)` 便可以返回一个类的实例，其中 args 的参数与 `__init__()` 方法中申明的参数一样。

`__init__()` 并不是真正意义上的构造方法，`__init__()` 方法所做的工作是在类的对象创建好之后进行变量的初始化。`__new__()` 方法才会真正创建实例，是类的构造方法。这两个方法都是 object 类中默认的方法，继承自 object 的新式类，如果不覆盖这两个方法将会默认调用 object 中对应的方法。

来看看 `__new__()` 方法和 `__init__()` 方法的定义：

- `object.__new__(cls[, args...])`：其中 cls 代表类，args 为参数列表
- `object.__init__(self[, args...])`：其中 self 代表实例对象，args 为参数列表

这两个方法之间有些不同点，总结如下：

- 根据 Python 文档可知，`__new__()` 方法是静态方法，而 `__init__()` 为实例方法
- `__new__()` 方法一般需要返回类的对象，当返回类的对象时将会自动调用 `__init__()` 方法进行初始化，如果没有对象返回，则 `__init__()` 方法不会被调用。`__init__()` 方法不需要显示返回，默认为 None，否则会在运行时抛出 TypeError
- 当需要控制实例创建的时候可使用 `__new__()` 方法，而控制实例初始化的时候使用 `__init__()` 方法
- 一般情况下不需要覆盖 `__new__()` 方法，但当子类继承自不可变类型，如 `str`、`int`、`unicode` 或者 `tuple` 的时候，往往需要覆盖该方法
- 当需要覆盖 `__new__()` 和 `__init__()` 方法的时候这两个方法的参数必须保持一致，如果不一致将导致异常。

一般情况下覆盖 `__init__()` 能满足大部分需求，特殊情况下需要覆盖 `__new__()` 方法：

- 当类继承（如 str、int、unicode、tuple 或者 forzenset 等）不可变类型且默认的 `__new__()` 方法不能满足需求的时候。比如需要一个不可修改的集合，该集合能够将任何以空格隔开的字符串变为集合中的元素：

```
class UserSet(frozenset):
    def __new__(cls, *args):
        if args and isinstance(args[0], basestring):
            args = (args[0].split(), ) + args[1:]
        return super(UserSet, cls).__new__(cls, *args)
```

- 用来实现工厂模式或者单例模式或者进行元类编程（元类编程中常常需要使用 `__new__()` 来控制对象创建）的时候。

  以简单工厂为例子，它由一个工厂类根据传入的参量决定创建出哪一种产品类的实例，属于类的创建型模式：

```
class Shape(object):
    def __init__(object):
        pass
    def draw(self):
        pass

class Triangle(Shape):
    def __init__(self):
        print("I am a triangle")
    def draw(self):
        print("I am drawing triangle")

class Rectangle(Shape):
    def __init__(self):
        print("I am a rectangle")
    def draw(self):
        print("I am drawing triangle")

class Trapezoid(Shape):
    def __init__(self):
        print("I am a trapezoid")
    def draw(self):
        print("I am drawing triangle")

class Diamond(Shape):
    def __init__(self):
        print("I am a diamond")
    def draw(self):
        print("I am drawing triangle")

class ShapeFactory(object):
    shapes = {"triangle": Triangle, "rectangle": Rectangle, "trapezoid": Trapezoid
, "diamond": Diamond}
    def __new__(class, name):
        if name in ShapeFactory.shapes.keys():
            print("creating a new shape {}".format(name))
            return ShapeFactory.shapes[name]()
        else:
            print("craeting a new shape {}".format(name))
            return Shape()
```

在 ShapeFactory 类中重新覆盖了 `__new__()` 方法，外界通过调用该方法来创建其所需的对象类型，但如果所请求的类是系统所不支持的，则返回 Shape 对象。在引入了工厂类之后，只需要使用如下形式就可以创建不同的图形对象：

```
ShapeFactory("rectangle").draw()
```

- 作为用来初始化的 `__init__()` 方法在多继承的情况下，子类的 `__init__()` 方法如果不显式调用父类的 `__init__()` 方法，则父类的 `__init__()` 方法不会被调用。

要显式调用父类的 `__init__()` 方法：`super(子类, self).__init__()`。对于多继承的情况，我们可以通过迭代子类的 `__bases__` 属性中的内容来逐一调用父类的初始化方法。

建议 56：理解名字查找机制

在 Python 中，所有所谓的变量，其实都是名字，这些名字指向一个或多个 Python 对象。

所有的这些名字，都存在于一个表里（又称为命名空间），一般情况下，我们称之为局部变量（locals），可以通过 `locals()` 函数调用看到。

在一个 `globals()` 的表里可以看到全局变量，注意如果是在 Python shell 中执行 `locals()`，也可以看到全局的变量。如果在一个函数里面定义这些变量，情况就会有所不同。

Python 中所有的变量名都是在赋值的时候生成的，而对任何变量名的创建、查找或者改变都会在命名空间（namespace）中进行。变量名所在的命名空间直接决定了其能访问到的范围，即变量的作用域。Python 中的作用域自 Python2.2 之后分为局部作用域（local）、全局作用域（Global）、嵌套作用域（enclosing functionas locals）以及内置作用域（Build-in）这 4 种。

- 局部作用域：一般来说函数的每次调用都会创建一个新的本地作用域，拥有新的命名空间。因此函数内的变量名可以与函数外的其他变量名相同，由于其命名空间不通过，并不会产生冲突。默认情况下函数内部任意的赋值操作（包括 = 语句、import 语句、def 语句、参数传递等）所定义的变量名，如果没用 global 语句，则申明都为局部变量，即仅在该函数内可见
- 全局作用域：定义在 Python 模块文件中的变量名拥有全局作用域，这里的全局仅限单个文件，即在一个文件的顶层的变量名仅在这个文件内可见，并非所有的文件，其他文件中想使用这些变量名必须先导入文件对应的模块。当在函数之外给一个变量赋值时是在其全局作用域的情况下进行的。
- 嵌套作用域：一般在多重函数嵌套的情况下才会注意到。需要注意的是 global 语句仅针对全局变量，在嵌套作用域的情况下，如果想在嵌套的函数内修改外层函数中定义的变量，即使使用 global 进行申明也不能达到目的，其结果最终是在嵌套的函数所在的命名空间中创建了一个新的变量。
- 内置作用域：它是通过一个标准库中名为 `__builtin__` 的模块来实现的

当访问一个变量的时候，其查找顺序遵循变量解析机制 LEGB 法则，即依次搜索 4 个作用域：局部作用域、嵌套作用域、全局作用域以及内置作用域，并在第一个找到的地方停止搜寻，如果没有搜到，则会抛出异常。具体来说 Python 的名字查找机制如下：

- 在最内层的范围内查找，一般而言，就是函数内部，即在 `locals()` 里面查找
- 在模块内查找，即在 `globals()` 里面查找
- 在外层查找，即在内置模块中查找，也就是在 `__builtin__` 中查找

在 CPython 的实现中，只要出现了赋值语句（或者称为名字绑定），那么这个名字就被当做局部变量来对待。需要改变全局变量时，使用 global 关键字。

在 Python 闭包中，有这样的问题：

```
>>> def foo():
    a = 1
    def bar():
        b = a * 2
        a = b + 1
        print(a)
    return bar

>>> foo()()
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    foo()()
  File "<pyshell#8>", line 4, in bar
    b = a * 2
UnboundLocalError: local variable 'a' referenced before assignment
```

在闭包 `bar()` 中，在编译代码为字节码时，因为存在 `a = b + 1` 这条语句，所以 a 被当做了局部变量看待，而执行时 `b = a * 2` 先执行，此时局部变量 a 尚不存在，所以就产生了一个异常。在 Python2.X 中可以使用 global 关键字解决部分问题，先把 a 创建为一个模块全局变量，然后在所有读写（包括只是访问）该变量的作用域中都要先使用 global 声明其为全局变量。

```
>>> a = 1
>>> def foo(x):
        global a
        a = a * x
        def bar():
            global a
            b = a * 2
            a = b + 1
            print(a)
        return bar
>>> foo(1)()
3
```

编程语言并不提倡全局变量，而且这种写法有时候还影响业务逻辑。此外，还有把 a 作为容器的一个元素来对待的方案，但也都相当复杂。真正的解决方案是 Python3 引入的 `nonlocal` 关键字：

```
>>> def foo(x):
    a = x
    def bar():
        nonlocal a
        b = a * 2
        a = b + 1
        print(a)
    return bar

>>> foo(1)
<function foo.<locals>.bar at 0x105de2bf8>
```

建议 57：为什么需要 self 参数

在类中当定义实例方法的时候需要将第一个参数显式声明为 self，而调用的时候并不需要传入该参数。

self 表示的就是实例对象本身，即类的对象在内存中的地址。self 是对对象本身的引用。我们在调用实例方法的时候也可以直接传入实例对象。其实 self 本身并不是 Python 的关键字（cls 也不是），可以将 self 替换成任何你喜欢的名称，如 this、obj 等，实际效果和 self 是一样的（但并不推荐，因为 self 更符合约定俗成的原则）

在方法声明的时候需要定义 self 作为第一个参数，而调用方法的时候却不用传入这个参数。虽然这并不影响语言本身的使用，而且也很容易遵循这个规则，但既然如此，为什么必须在定义方法的时候声明 self 参数？原因如下：

- Python 在当初设计的时候借鉴了其他语言的一些特征，如 `Moudla-3` 中方法会显式地在参数列表中传入 self。Python 起源于 20 世纪 80 年代末，那个时候的很多语言都有 self，如 `Smalltalk` 、`Modula-3` 等。Python 在最开始设计的时候受到了其他语言的影响，因此借鉴了其中的一些理念。

- Python 语言本身的动态性决定了使用 self 能够带来一定便利。

  Python 属于一级对象语言（first class object)，如果 m 是类 A 的一个方法，有好几种方式都可以引用该方法：

  ```
  >>> class A:
          def m(self, value):
              pass
  >>> A.__dict__["m"]
  >>> A.m.__func__
  ```

  实例方法是作用于对象的，最简单的方式就是将对象本身传递到该方法中去，self 的存在保证了 `A.__dict__['m'](a, 2)` 的使用和 `a.(2)` 一致。同时当子类覆盖了父类中的方法但仍然想调用该父类的方法的时候，可以方便地使用 `baseclass.methodname(self,`

`<argument list>)` 或 `super(childclass, self).methodname(<argument list>)` 来实现。

- 在存在同名的局部变量以及实例变量的情况下使用 self 使得实例变量更容易被区分

Guido 认为，基于 Python 目前的一些特性（如类中动态添加方法，在类风格的装饰器中没有 self 无法确认是返回一个静态方法还是类方法等）保留其原有设计是个更好的选择，更何况 Python 的哲学是：显示优于隐式（Explicit is better than implicit）。

建议 58：理解 MRO 与多继承

Python 也支持多继承，语法：

```
class DerivedClassName(Base1, Base2, Base3)
```

古典类和新式类之间所采用的 MRO（Method Resolution Order，方法解析顺序）实现方式存在差异。

在古典类中，MRO 搜索采用简单的自左向右的深度优先方法，即按照多继承申明的顺序形成继承树结构，自顶向下采用深度优先的搜索顺序，当找到所需要的属性或者方法的时候就停止搜索。

而新式类采用的而是 C3 MRO 搜索方法，该算法描述如下：

> 假定，C1C2...CN 表示类 C1 到 CN 的序列，其中序列头部元素（head）=C1，序列尾部（tail）定义 = C2...CN；
>
> C 继承的基类自左向右分别表示为 B1，B2...BN
>
> L[C] 表示 C 的线性继承关系，其中 L[object] = object。
>
> 算法具体过程如下：
>
> L[C(B1...BN)] = C + merge(L[B1] ... L[BN], B1 ... BN)
>
> 其中 merge 方法的计算规则如下：在 L[B1]...L[BN]，B1...BN 中，取 L[B1] 的 head，如果该元素不在 L[B2]...L[BN]，B1...BN 的尾部序列中，则添加该元素到 C 的线性继承序列中，同时将该元素从所有列表中删除（该头元素也叫 good head），否则取 L[B2] 的 head。继续相同的判断，直到整个列表为空或者没有办法找到任何符合要求的头元素（此时，将引发一个异常）。

关于 MRO 的搜索顺序也可以在新式类中通过查看 `__mro__` 属性得到证实。

实际上 MRO 虽然叫方法解析顺序，但它不仅是针对方法搜索，对于类中的数据属性也适用。

菱形继承是我们在多继承设计的时候需要尽量避免的一个问题。

建议 59：理解描述符机制

除了在不同的局部变量、全局变量中查找名字，还有一个相似的场景，那就是查找对象的属性。在 Python 中，一切皆是对象，所以类也是对象，类的实例也是对象。

每一个类都有一个 `__dict__` 属性，其中包含的是它的所有属性，又称为类属性。

除了与类相关的类属性之外，每一个实例也有相应的属性表（ `__dict__` ），称为实例属性。当我们通过实例访问一个属性时，它首先会尝试在实例属性中查找，如果找不到，则会到类属性中查找。

实例可以访问类属性，但与读操作有所不同，如果通过实例增加一个属性，只能改变此实例的属性，对类属性而言，并没有变化。

能不能给类增加一个属性？答案是，能，也不能。说能是因为每一个 class 也是一个对象，动态地增减对象的属性与方法正是 Python 这种动态语言的特性，自然是支持的。

说不能，是因为在 Python 中，内置类型和用户定义的类型是有分别的，内置类型并不能够随意地为它增加属性或方法。

当我们通过 "." 操作符访问一个属性时，如果访问的实例属性，与直接通过 `__dict__` 属性获取相应的元素是一样的；而如果访问的是类属性，则并不相同；"." 操作符封装了对两种不同属性进行查找的细节。

访问类属性时，通过 `__dict__` 访问和使用 "." 操作符访问是一样的，但如果是方法，却又不是如此了。

当通过 "." 操作符访问时，Python 的名字查找并不是先在实例属性中查找，然后再在类属性中查找那么简单，实际上，根据通过实例访问属性和根据类访问属性的不同，有以下两种情况：

- 一种是通过实例访问，比如代码 `obj.x` ，如果 x 是一个描述符，那么 `__getattribute__()` 会返回 `type(obj).__dict__['x'].__get__(obj, type(obj))` 结果，即： `type(obj)` 获取 obj 的类型； `type(obj).__dict__['x']` 返回的是一个描述符，这里有一个试探和判断的过程；最后调用这个描述符的 `__get__()` 方法。
- 另一个是通过类访问的情况，比如代码 `cls.x` ，则会被 `__getattribute__()` 转换为 `cls.__dict__['x'].__get__(None, cls)` 。

描述符协议是一个 Duck Typing 的协议，而每一个函数都有 `__get__` 方法，也就是说其他每一个函数都是描述符。

描述符机制有什么作用？其实它的作用编写一般程序的话还真用不上，但对于编写程序库的读者来说就有用了，比如已绑定方法和未绑定方法。

由于对描述符的 `__get__()` 的调用参数不同，当以 `obj.x` 的形式访问时，调用参数是 `__get__(obj, type(obj))` ；而以 `cls.x` 的形式访问时，调用参数是 `__get__(None, type(obj))` ，这可以通过未绑定方法的 `im_self` 属性为 None 得到印证。

除此之外，所有对属性、方法进行修饰的方案往往都用到了描述符，比如 `classmethod` 、 `staticmethod` 和 `property` 等。以下是 property 的参考实现：

```
class Property(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        self.__doc__ = doc
    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError, "unreadable attribute"
        return self.fget(obj)
    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError, "can't set attribute"
        self.fset(obj, value)
    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError, "can't delete attribute"
        self.fdel(obj)
```

建议 60：区别 `__getattr__()` 和 `__getattribute__()` 方法

`__getattr__()` 和 `__getattribute__()` 都可以用作实例属性的获取和拦截（仅对实例属性（instance varibale）有效，非类属性），`__getattr__()` 适用于未定义的属性，即该属性在实例中以及对应的类的基类以及祖先类中都不存在，而 `__getattribute__()` 对于所有属性的访问都会调用该方法。

需要注意的是 `__getattribute__()` 仅用于新式类。

当访问一个不存在的实例属性的时候就会抛出 `AttributeError` 异常。这个异常时由内部方法 `__getattribute__(self, name)` 抛出的，因为 `__getattribute__()` 会被无条件调用，也就是说只要涉及实例属性的访问就会调用该方法，它要么返回实际的值，要么抛出异常。Python 的文档中也提到了这一点。

实际上 `__getattr__()` 方法仅如下情况才被调用：属性不在实例的 `__dict__` 中；属性不在其基类以及祖先类的 `__dict__` 中；触发 `AttributeError` 异常时（注意，不仅仅是 `__getattribute__()` 方法的 `AttributeError` 异常，property 中定义的 `get()` 方法抛出异常的时候也会调用该方法）。

当这两个方法同时被定义的时候，要么在 `__getattribute__()` 中显式调用，要么触发 `AttributeError` 异常，否则 `__getattr__()` 永远不会被调用。`__getattribute__()` 及 `__getattr__()` 方法都是 Object 类中定义的默认方法，当用户需要覆盖这些方法时有以下几点注意事项：

- 避免无穷递归。比如覆盖 `__getattribute__()` 时使用了 `self.__dict__[attr]`，正确的做法是使用 `super(obj, self).__getattribute__(attr)` 或者 `object.__getattribute(self, attr)`。无穷递归是覆盖 `__getattr__()` 和 `__getattribute__()` 方法的时候需要特别小心
- 访问未定义的属性。如果在 `__getattr__()` 方法中不抛出 `AttributeError` 异常或者显式返回一个值，则会返回 None，此时可能会影响到程序的实际运行预期。

另外关于 `__getattr__()` 和 `__getattribute__()` 有以下两点提醒：

- 覆盖了 `__getattribute__()` 方法之后，任何属性的访问都会调用用户定义的 `__getattribute__()` 方法，性能上会有所损耗，比使用默认的方法要慢。
- 覆盖的 `__getattr__()` 方法如果能够动态处理事先未定义的属性，可以更好地实现数据隐藏。因为 `dir()` 通常只显示正常的属性和方法，因此不会将该属性列为可用属性。
- property 也能控制属性的访问，如果一个类中同时定义了 `property`、`__getattribute__()` 和 `__getattr__()` 来对属性进行访问控制，则最先搜索的是 `__getattribute__()` 方法，然而由于 property 对象并不存在 dict 中，因此并不能返回该方法，此时会搜索 property 中定义的 `get()` 方法。当用 `property` 中的 `set()` 方法进行修改并再次访问 `property` 的 `get()` 方法时会抛出异常，这种情况下会触发对 `__getattr__()` 方法的调用。对类变量的访问不会涉及 `__getattribute__()` 和 `__getattr__()` 方法。

建议 61：使用更为安全的 property

property 是用来实现属性可管理性的 `built-in` 数据类型（注意：很多地方将 property 称为函数，然而它实际上是一种实现了 `__get__()`、`__set__()` 方法的类，用户也可以根据自己的需要定义个性化的 property），其实质是一种特殊的数据描述符（数据描述符：如果一个对象同时定义了 `__get__()` 和 `__set__()` 方法，则称为数据描述符，如果仅定义了 `__get__()` 方法，则称为非数据描述符）。它和普通描述符的区别在于：普通描述符提供的是一种较为低级的控制属性访问的机制，而 property 是它的高级应用，它以标准库的形式提供描述符的实现，其签名形式为：

```
property(fget=None, fset=None, fdel=None, doc=None) -> property attribute
```

Property 常见的使用形式有以下几种：

- 第一种形式如下：

```
class Some_Class(object):
    def __init__(self):
        self._somevalue = 0
    def get_value(self):
        return self._somevalue
    def set_value(self, value):
        self._somevalue = value
    def del_attr(self):
        del self._somevalue
    x = property(get_value, set_value, del_attr, "I am the ''x' property.")
```

- 第二种形式如下：

```
class Some_Class(object):
    _x = None
    def __init__(self):
        self._x = None
    @property
    def x(self):
        return self._x
    @x.setter
    def x(self, value):
        self._x = value
    @x.deleter
    def x(self):
        del self._x
```

property 的优势可以简单地概括为以下几点：

- 代码更简洁，可读性更强。
- 更好的管理属性的访问。property 将对属性的访问直接转换为对对应的 get、set 等相关函数的调用，属性能够更好地被控制和管理，常见的应用场景如设置校验、检查赋值的范围以及对某个属性进行二次计算之后再返回给用户或者计算某个依赖于其他属性的属性。 创建一个 property 实际上就是将其属性的访问与特定的函数关联起来，相对于标准属性的访问，property 的作用相当于一个分发器，对某个属性的访问并不直接操作具体的对象，而对标准属性的访问没有中间这一层，直接访问存储属性的对象。
- 代码可维护性更好。property 对属性进行再包装，以类似于接口的形式呈现给用户，以统一的语法来访问属性，当具体实现需要改变的时候，访问的方式仍然可以保持一致
- 控制属性访问权限，提高数据安全性。如果用户想设置某个属性为只读：

```
class PropertyTest(object):
    def __init__(self):
        self.__var1 = 20
    @property
    def x(self):
        return self.__var1
```

> 值得注意的是，使用 property 并不能真正完全达到属性只读的目的，正如以双下划线命令的变量并不是真正的私有变量一样，这些方法只是在直接修改属性这条道路上增加了一些障碍。

property 本质并不是函数，而是特殊类，既然是类的话，那么就可以被继承，因此用户便可以根据自己的需要定义 property：

```
def update_meta(self, other):
    self.__name__ = other.__name__
    self.__doc__ = other.__doc__
    self.__dict__.update(other.__dict__)
    return self

class UserProperty(property):
    def __new__(cls, fget=None, fset=None, fdel=None, doc=None):
        if fget is not None:
            def __get__(obj, objtype=None, name=fget.__name__):
                fegt = getattr(obj, name)
                return fget()
            fget = update_meta(__get__, fget)

        if fset is not None:
            def __set__(obj, value, name=fset.__name__):
                fset = getattr(obj, name)
                return fset(value)
            fset = update_meta(__set__, fset)

        if fdel is not None:
            def __delete__(obj, name=fdel.__name__):
                fdel = getattr(obj, name)
                return fdel()
            fdel = update_meta(__delete__, fdel)
        return property(fget, fset, fdel, doc)

class C(object):
    def get(self):
        return self._x
    def set(self, x):
        self._x = x
    def delete(self):
        del self._x
    x = UserProperty(get, set, delete)
```

上例中 UserProperty 继承自 property，其构造函数 `__new__(cls, fget=None, fset=None, fdel=None, doc=None)` 中重新定义了 `fget()`、`fset()` 以及 `fdel()` 方法以满足用户特定的需要，最后返回的对象实际还是 property 的实例，因此用户能够像使用 property 一样使用 UserProperty。

使用 property 并不能真正完全达到属性只读的目的，用户仍然可以绕过阻碍来修改变量。真正实现只读属性的可行实现：

```
def ro_property(obj, name, value):
    setattr(obj.__class__, name, property(lambda obj: obj.__dict__["__" + name]))
    setattr(obj, "__" + name, value)

class ROClass(object):
    def __init__(self, name, available):
        ro_property(self, "name", name)
        self.available = available

a = ROClass("read only", True)
print(a.name)
a._Article__name = "modify"
print(a.__dict__)
print(ROClass.__dict__)
print(a.name)
```

建议 62：掌握 metaclass

什么是元类？

- 元类是关于类的类，是类的模版
- 元类是用来控制如何创建类的，正如类是创建对象的模版一样
- 元类的实例为类，正如类的实例为对象

当使用关键字 class 的时候，Python 解释器在执行的时候就会创建一个对象（这里的对象是指类而非类的实例）

既然类是对象，那么它就有其所属的类型，也一定还有什么东西能够控制它的生成。通过 type 查看会发现 UserClass 的类型为 type，而其对象 UserClass() 的类型为类 A。

同时我们知道 type 还可以这样使用：

`type(类名，父类的元组（针对继承的情况，可以为空），包含属性的字典（名称和值））`

type 通过接受类的描述符作为参数返回一个对象，这个对象可以被继承，属性能够被访问，它实际是一个类，其创建由 type 控制，由 type 所创建的对象的 `__class__` 属性为 type。type 实际上是 Python 的一个内建元类，用来指导类的生成。当然，除了使用内建元类 type，用户也可以通过继承 type 来自定义元类。

利用元类实现强制类型检查：

```
class TypeSetter(object):
    def __init__(self, fieldtype):
        self.fieldtype = fieldtype
    def is_valid(self, value):
        return isinstance(value, self.fieldtype)
class TypeCheckMeta(type):
    def __new__(cls, name, bases, dict):
        return super(TypeCheckMeta, cls).__new__(cls, name, bases, dict)
    def __init__(cls, name, bases, dict):
        cls._fields = {}
        for key, value in dict.items():
            if isinstance(value, TypeSetter):
                cls._fields[key] = value
  def sayHi(cls):
      print("Hi")
```

TypeSetter 用来设置属性的类型，TypeCheckMeta 为用户自定义的元类，覆盖了 type 元类中的 `__new__()` 方法和 `__init__()` 方法，虽然也可以直接使用 `TypeCheckMeta(name, bases, dict)` 这种方式来创建类，但更为常见的是在需要被生成的类中设置 `__metaclass__` 属性，两种用法是等价的：

```
class TypeCheck(object):
    __metaclass__ = TypeCheckMeta
    def __setattr__(self, key, value):
        if key in self._fields:
            if not self._fields[key].is_valid(value):
                raise TypeError("Invalid type for field")
            super(TypeCheck, self).__setattr__(key, value)

class MetaTest(TypeCheck):
    name = TypeSetter(str)
    num = TypeSetter(int)

mt = MetaTest()
mt.name = "apple"
mt.num = "test"
```

当类中设置了 `__metaclass__` 属性的时候，所有继承自该类的子类都将使用所设置的元类来指导类的生成。

实际上，在新式类中当一个类未设置 `__metaclass__` 属性的时候，它将使用默认的 `type` 元类来生成类。而当该属性被设置时查找规则如下：

- 如果存在 `dict["__metaclass__"]`，则使用对应的值来构建类；否则使用其父类 `dict["__metaclass__"]` 中所指定的元类来构建类，当父类中也不存在指定的 `metaclass` 的情形下使用默认元类 type。

- 对于古典类，条件 1不满足的情况下，如果存在全局变量 `__metaclass__` ，则使用该变量所对应的元类来构建类；否则使用 `type.ClassType` 。

需要额外提醒的是，元类中所定义的方法为其所创建的类的类方法，并不属于该类的对象。比如上例中的 `mt.sayHi()` 会抛出异常，正确调用方法为： `MetaTest.sayHi()` 。

什么情况下会用到元类？有句话是这么说的：当你面临一个问题还在纠结要不要使用元类的时候，往往会有其他的更为简单的解决方案。

几个使用元类的场景：

- 利用元类来实现单例模式：

```
class Singleton(type):
    def __init__(cls, name, bases, dic):
        super(Singleton, cls).__init__(name, bases, dic)
        cls.instance = None
    def __call__(cls, *args, **kwargs):
        if cls.instance is None:
            cls.instance = super(Singleton, cls).__call__(*args, **kwargs)
        else:
            print("warning: only allowed to create one instance, minstance already
 exists!")
        return cls.instance

class MySingleton(object):
    __metaclass__ = Singleton
```

- 第二个例子来源于 Python 的标准库 string.Template.string，它提供简单的字符串替换功能。 `Template("$name $age").substitute({"name":"admin"}, age=26)`

  该标准库的源代码中就用到了元类， `Template` 的元类为 `_TemplateMetaclass` 。 `_TemplateMetaclass` 的 `__init__()` 方法通过查找属性（pattern、delimiter 和 idpattern）并将其构建为一个编译好的正则表达式存放在 pattern 属性中。用户如果需要自定义分隔符（delimiter）可以通过继承 Template 并覆盖它的类属性 delimiter 来实现。

  另外在 Django ORM、AOP 编程中也有大量使用元类的情形。

谈谈关于元类需要注意的几点：

- 区别类方法与元方法（定义在元类中的方法）。元方法可以从元类或者类中调用，而不能从类的实例中调用；但类方法可以从类中调用，也可以从类的实例中调用
- 多继承需要严格限制，否则会产生冲突。因为 Python 解释器并不知道多继承的类是否兼容，因此会发出冲突警告。解决冲突的办法是重新定义一个派生的元类，并在要集成的类中将其 `__metaclass__` 属性设置为该派生类。

建议 63：熟悉 Python 对象协议

因为 Python 是一门动态语言，Duck Typing 的概念遍布其中，所以其中的 Concept 并不以类型的约束为载体，而另外使用称为协议的概念。在 Python 中就是我需要调用你某个方法，你正好就有这个方法。比如在字符串格式化中，如果有占位符 %s，那么按照字符串转换的协议，Python 会自动地调用相应对象的 `__str__()` 方法。

除了 `__str__()` 外，还有其他的方法，比如 `__repr__()`、`__init__()`、`__long__()`、`__float__()`、`__nonzero__()` 等，统称类型转换协议。除了类型转换协议之外，还要许多其他协议。

- 用以比较大小的协议，这个协议依赖于 `__cmp__()`，与 C 语言库函数 cmp 类似，当两者相等时，返回 0，当 `self < other` 时返回负值，反之返回正值。因为这种复杂性，所以 Python 又有 `__eq__()`、`__ne__()`、`__lt__()`、`__gt__()` 等方法来实现相等、不等、小于和大于的判定。这也就是 Python 对 `==`、`!=`、`<` 和 `>` 等操作符的进行重载的支撑机制。
- 数值类型相关的协议，这一类的函数比较多。基本上，只要实现了那么几个方法，基本上就能够模拟数值类型了。不过还需要提到一个 Python 中特有的概念：反运算。类似 `__radd__()` 的方法，所有的数值运算符和位运算符都是支持的，规则也是一律在前面加上前缀 r 即可。

- 容器类型协议。容器的协议是非常浅显的，既然为容器，那么必然要有协议查询内含多少对象，在 Python 中，就是要支持内置函数 `len()`，通过 `__len__()` 来完成。而 `__getitem__()`、`__setitem__()`、`__delitem__()` 则对应读、写和删除，也很好理解。`__iter__()` 实现了迭代器协议，而 `__reversed__()` 则提供对内置函数 `reversed()` 的支持。容器类型中最有特色的是对成员关系的判断符 in 和 not in 的支持，这个方法叫 `__contains__()`，只要支持这个函数就能够使用 in 和 not in 运算符了。

- 可调用对象协议。所谓可调用对象，即类似函数对象，能够让类实例表现得像函数一样，这样就可以让每一个函数调用都有所不同。

```
class Functor(object):
    def __init__(self, context):
        self._context = context
    def __call__(self):
        print("do something with {}".format(self._context))
lai_functor = Functor("lai")
yong_functor = Functor("yong")
lai_functor()
yong_functor()
```

- 与可调用对象差不多的，还有一个可哈希对象，它是通过 `__hash__()` 方法来支持 `hash()` 这个内置函数的，这在创建自己的类型时非常有用，因为只有支持可哈希协议的类型才能作为 dict 的键类型（不过只要继承自 object 的新式类就默认支持了）

- 描述符协议和属性交互协议（`__getattr__()`、`__setattr__()`、`__delattr__()`），还有上下文管理器协议，也就是对 with 语句的支持，这个协议通过 `__enter__()` 和 `__exit__()` 两个方法来实现对资源的清理，确保资源无论在什么情况下都会正常清理。

协议不像 C++、Java 等语言中的接口，它更像是声明，没有语言上的约束力。

建议 64：使用操作符重载实现中缀语法

模拟 C++ 的流输出，是一种对特性的滥用，不应提倡。

管道的处理非常清晰，因为它是中缀语法，而我们常用的 Python 是前缀语法，比如类似的 Python 代码应该是 `sort(ls(), reverse=True)`。

管道符号在 Python 中，也是或符号，由 Julien Palard 开发了一个 pipe 库，这个 pipe 库的核心代码只有几行，就是重载了 `__ror__()` 方法：

```
class Pipe:
    def __init__(self, function):
        self.function = function
    def __ror__(self, other):
        return self.function(other)
    def __call__(self, *args, **kwargs):
        return Pipe(lambda x: self.function(x, *args, **kwargs))
```

这个 Pipe 类可以当成函数的 decorator 来使用：

```
@Pipe
def where(iterable, predicate):
    return (x for x in iterable if (predicate(x)))
```

`pipe` 库内置了一堆这样的处理函数，比如 `sum`、`select`、`where` 等函数尽在其中：

```
fib() | take_while(lambda x: x < 1000000) \
      | where(lambda x: x % 2) \
      | select(lambda x: x * x) \
      | sum()
```

这段代码就是找出小于 1000000 的斐波那契数，并计算其中的偶数的平方之和。

此外，pipe 是惰性求值的，所以我们完全可以弄一个无穷生成器而不用担心内存被用完。

除了处理数值很方便，用它来处理文本也一样简单。比如读取文件，统计文件中每个单词出现的次数，然后按照次数从高到低对单词排序：

```
from __future__ import print_function
from re import split
from pipe import *
with open("test_descriptor.py") as f:
    print(f.read()
          | Pipe(lambda x: split("/W+", x))
          | Pipe(lambda x:(i for i in x if i.strip()))
          | groupby(lambda x:x)
          | select(lambda x:(x[0], (x[1] | count)))
          | sort(key=lambda x: x[1], reverse=True)
         )
```

建议 65：熟悉 Python 的迭代器协议

Python 的迭代器集成在语言之中，不像 C++ 中那样需要专门去理解这一个概念。

但是，并非所有的时候都能够隐藏细节。首先介绍一下 `iter()` 函数，`iter()` 可以输入两个实参，第二个可选参数可以忽略。`iter()` 函数返回一个迭代器对象，接受的参数是一个实现了 `__iter__()` 方法的容器或迭代器（精确来说，还支持仅有 `__getitem__()` 方法的容器）。对于容器而言，`__iter__()` 方法返回一个迭代器对象，而对迭代器而言，它的 `__iter__()` 方法返回其自身。

迭代器协议，所谓协议，是一种松散的约定，并没有相应的接口定义，所以把协议简单归纳如下：

- 实现 `__iter__()` 方法，返回一个迭代器
- 实现 `next()` 方法，返回当前的元素，并指向下一个元素的位置，如果当前位置已无元素，则抛出 `StopIteration` 异常。

其实 for 语句就是对获取容器的迭代器、调用迭代器的 `next()` 方法以及对 `StopIteration` 进行处理等流程进行封装的语法糖（类似的语法糖还有 `in/not in` 语句）。

迭代器最大的好处是定义了统一的访问容器（或集合）的统一接口，所以程序员可以随时定义自己的迭代器，只要实现了迭代器协议即可。除此之外，迭代器还有惰性求值的特性，它仅可以在迭代至当前元素时才计算（或读取）该元素的值，在此之前可以不存在，在此之后也可以销毁，也就是说不需要在遍历之前实现准备好整个迭代过程中的所有元素，所以非常适合遍历无穷个元素的集合或或巨大的事物（斐波那契数列、文件）。

迭代器在一些应用场景更省 CPU 计算资源，所以在编写代码中应当多多使用迭代器协议，避免劣化代码。从 Python2.3 版本开始，`itertools` 成为了标准库的一员已经充分印证这个观点。

`itertools` 的目标是提供一系列计算快速、内存高效的函数，这些函数可以单独使用，也可以进行组合，这个模块受到了 Haskell 等函数式编程语言的启发，所以大量使用 `itertools` 模块中的函数的代码，看起来有点像函数式编程语言。比如 `sum(imap(operator.mul, vector1,`

`vector2))` 能够用来运行两个向量的对应元素乘积之和。

`itertools` 最为人所熟知的版本，应该算是 zip、map、filter、slice 的替代，`izip（izip_longest）`、`imap（startmap）`、`ifilter（ifilterfalse）`、`islice`，它们与原来的那几个内置函数有一样的功能，只是返回的是迭代器（在 Python3 中，新的函数彻底替换掉了旧函数）

除了对标准函数的替代，`itertools` 还提供了以下几个有用的函数：`chain()` 用以同时连续地迭代多个序列；`compress()`、`dropwhile()` 和 `takewhile()` 能用遴选序列元素；`tee()` 就像同名的 UNIX 应用程序，对序列作 n 次迭代；而 `groupby` 的效果类似 SQL 中相同拼写的关键字所带的效果。

```
[k for k, g in groupby("AAAABBBCCDAABB")] --> A B C D A B
[list(g) for k, g in groupby("AAAABBBCCD")] --> AAAA BBB CC D
```

除了这些针对有限元素的迭代帮助函数之外，还有 `count()`、`cycle()`、`repeat()` 等函数产生无穷序列，这 3 个函数就分别可以产生算术递增数列、无限重复实参的序列和重复产生同一个值的序列。

还有几个组合函数：

- `product()`：计算 m 个序列的 n 次笛卡尔积
- `permutations()`：产生全排列
- `combinations()`：产生无重复元素的组合
- `combinations_with_replacement()`：产生有重复元素的组合

```
>>> list(product("ABCD", repeat=2))
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'A'), ('B', 'B'), ('B', 'C'), (
'B', 'D'), ('C', 'A'), ('C', 'B'), ('C', 'C'), ('C', 'D'), ('D', 'A'), ('D', 'B'), ('D'
, 'C'), ('D', 'D')]
>>> list(permutations("ABCD", 2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'A'), ('B', 'C'), ('B', 'D'), ('C', 'A'), (
'C', 'B'), ('C', 'D'), ('D', 'A'), ('D', 'B'), ('D', 'C')]
>>> list(combinations("ABCD", 2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]
>>> list(combinations_with_replacement("ABCD", 2))
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'B'), ('B', 'C'), ('B', 'D'), (
'C', 'C'), ('C', 'D'), ('D', 'D')]
>>> for i in product("ABC", "123", repeat=2):
    print("".join(i))

# product() 可以接受多个序列
A1A1
A1A2
A1A3
A1B1
....
```

建议 66：熟悉 Python 的生成器

生成器，就是按一定的算法生成一个序列。迭代器虽然在某些场景表现得像生成器，但它绝非生成器；反而是生成器实现了迭代器协议的，可以在一定程度上看作迭代器。

如果一个函数，使用了 yield 语句，那么它就是一个生成器函数。当调用生成器函数时，它返回一个迭代器，不过这个迭代器是以生成器对象的形式出现的，这个对象带有 `__iter__()` 和 `next()` 方法。

每一个生成器函数调用之后，它的函数并不执行，而是到第一次调用 `next()` 的时候才开始执行。

当第一次调用 `next()` 方法时，生成器函数开始执行，执行到 yield 表达式为止。

直率地说， `send()` 方法很绕，其实 `send()` 是全功能版本的 `next()` ，或者说 `next()` 是 `send()` 的快捷方式，相当于 `send(None)` 。yield 表达式有一个返回值， `send()` 方法的作用就是控制这个返回值，使得 `yield` 表达式的返回值是它的实参。

除了能 yield 表达式的“返回值”之外，也可以让它抛出异常，这就是 `throw()` 方法的能力。对于常规业务逻辑的代码来说，对特定的异常有很好的处理（比如将异常信息写入日志后优雅的返回），从而实现从外部影响生成器内部的控制流。

当调用 `close()` 方法时，yield 表达式就抛出 `GeneratorExit` 异常，生成器对象会自行处理这个异常。当调用 `close()` 方法，再次调用 `next()`、`send()` 会使生成器对象抛出 `StopIteration` 异常。换言之，这个生成器对象已经不再可用。当生成器对象被 GC 回收时，会自动调用 `close()`。

生成器还有两个很棒的用处，其中之一就是实现 with 语句的上下文管理协议，利用的是调用生成器函数时函数体并不执行，当第一次调用 `next()` 方法时才开始执行，并执行到 yield 表达式后中止，直到下一次调用 `next()` 方法这个特性；其二是实现协程，利用的是 `send()`、`throw()`、`close()` 等特性。

上下文管理器协议，其实就是要求类实现 `__enter__()` 和 `__exit__()` 方法，但是生成器对象并没有这两个方法，所以 `contextlib` 提供了 `contextmanager` 函数来适配这两种协议：

```
from contextlib import contextmanager
@contextmanager
def tag(name):
    print("<{}>".format(name))
    yield
    print("</{}>".format(name))
>>> with tag("h1"):
    print("foo")
<h1>
foo
</h1>
```

通过 `contextmanager` 对 `next()`、`throw()`、`close()` 的封装，yield 大大简化了上下文管理器的编程复杂度，对提高代码可维护性有着极大的意义。除此之外，`yield` 和 `contextmanager` 也可以用以“池”模式中对资源的管理和回收。

建议 67：基于生成器的协程及 greenlet

协程，又称微线程和纤程等，据说源于 Simula 和 Modula-2 语言，现代编程语言基本上都支持这个特性，比如 Lua 和 ruby 都有类似的概念。协程往往实现在语言的运行时库或虚拟机中，操作系统对其存在一无所知，所以又被称为用户空间线程或绿色线程。又因为大部分协程的实现是协作式而非抢占式的，需要用户自己去调度，所以通常无法利用多核，但用来执行协作式多任务非常合适。用协程来做的东西，用线程或进程通常也是一样可以做的，但往往多了许多加锁和通信的操作。

基于生产着消费者模型，比较抢占式多线程编程实现和协程编程实现。线程实现至少有两点硬伤：

- 对队列的操作需要有显式/隐式（使用线程安全的队列）的加锁操作。
- 消费者线程还要通过 sleep 把 CPU 资源适时地“谦让”给生产者线程使用，其中的适时只能静态地使用经验值。

而使用协程可以比较好地解决：

```
# 队列容器
q = new queue
# 生产者协程
loop
    while q is not full
        create some new items
        add the items to q
    yield to consume
# 消费者协程
loop
    while q is not empty
        remove some items from q
        use the items
    yield to produce
```

但是这样做，损失了利用多核 CPU 的能力。

具体的生成器函数代码：

```
def consumer():
    while True:
        line = yield
        print(line.upper())
def producter():
    with open("/var/log/apache2/error_log", "r") as f:
        for i, line in enumerate(f):
            yield line
            print("processed line {}".format(i))
c = consumer()
c.next()
for line in producter():
    c.send(line)
```

协程，每输出一行大写的文字后都有一行来自主程序的处理信息，不会像抢占式的多线程程序那样"乱序"。Python2.X 版本的生成器无法实现所有的协程特性，是因为缺乏对协程之间复杂关系的支持。比如一个 yield 协程依赖另一个 yield 协程，且需要由最外层往最内层进行传值的时候，就没有解决办法。

这个问题直到 Python3.3 增加了 `yield from` 表达式以后才得以解决，通过 `yield from`，外层的生成器在接收到 `send()` 或 `throw()` 调用时，能够把实参直接传入内层生成器。

因为 Python2.x 版本对协程的支持有限，而协程又是非常有用的特性，所以很多 Pythonista 就开始寻求语言之外的解决方案，并编写了一系列的程序库，其中最受欢迎的是 greenlet。

greenlet 是一个 C 语言编写的程序库，它与 yield 关键字没有密切的关系。greenlet 这个库里最为关键的一个类型就是 PyGreenlet 对象，它是一个 C 结构体，每一个 PyGreenlet 都可以看到一个调用栈，从它的入口函数开始，所有的代码都在这个调用栈上运行。它能够随时记录代码运行现场，并随时中止，以及恢复。它跟 yield 所能够做到的相似，但更好的是它提供从一个 PyGreenlet 切换到另一个 PyGreenlet 的机制。

协程虽然不能充分利用多核，但它跟异步 I/O 结合起来以后编写 I/O 密集型应用非常容易，能够在同步的代码表面下实现异步的执行，其中的代表当属将 greenlet 与 libevent/libev 结合起来的 gevent 程序库，它是 Python 网络编程库。最后，以 gevent 并发查询 DNS 的例子为例，使用它进行并发查询 n 个域名，能够获得几乎 n 倍的性能提升：

```
import gevent
from gevent import socket
urls = ["www.google.com", "www.example.com", "www.python.org"]
jobs = [gevent.spawn(socket.gethostbyname, url) for url in urls]
gevent.joinall(jobs, timeout=2)
print([job.value for job in jobs])
```

建议 68：理解 GIL 的局限性

多线程 Python 程序运行的速度比只有一个线程的时候还要慢，除了程序本身的并行性之外，很大程度上与 GIL 有关。由于 GIL 的存在，多线程编程在 Python 中并不理想。GIL 被称为全局解释器锁（Global Interpreter Lock），是 Python 虚拟机上用作互斥线程的一种机制，它的作用是保证任何情况下虚拟机中只会有一个线程被运行，而其他线程都处于等待 GIL 锁被释放的状态。不管是在单核系统还是多核系统中，始终只有一个获得了 GIL 锁的线程在运行，每次遇到 I/O 操作便会进行 GIL 锁的释放。

但如果是纯计算的程序，没有 I/O 操作，解释器则会根据 sys.setcheckinterval 的设置来自动进行线程间的切换，默认情况下每隔 100 个时钟（这里的时钟指的是 Python 的内部时钟，对应于解释器执行的指令）就会释放 GIL 锁从而轮换到其他线程的执行。

在单核 CPU 中，GIL 对多线程的执行并没有太大影响，因为单核上的多线程本质上就是顺序执行的。但对于多核 CPU，多线程并不能真正发挥优势带来效率上明显的提升，甚至在频繁 I/O 操作的情况下由于存在需要多次释放和申请 GIL 的情形，效率反而会下降。

鉴于 Python 中对象的管理与引用计数器，在 Python 解释器中引入了 GIL，以保证对虚拟机内部共享资源访问的互斥性。GIL 的引入确实使得多线程不能再多核系统中发挥优势，但它也带来了一些好处：大大简化了 Python 线程中共享资源的管理，在单核 CPU 上，由于其本质是顺序执行的，一般情况下多线程能够获得较好的性能。此外，对于扩展的 C 程序的外部调用，即使其不是线程安全的，但由于 GIL 的存在，线程会阻塞直到外部调用函数返回，线程安全不再是一个问题。

针对 Python1.5，Greg Stein 发布了一个补丁，该补丁中 GIL 被完全移除，使用高粒度的锁来代替，然而多核多线程速度的提升并没有随着核数的增加而线性增长，反而给单线程程序的执行速度带来了一定的代价，速度大约降低了 40%。在 Python3.2 中重新实现了 GIL，其实现机制主要集中在两个方面：一方面是使用固定的时间而不是固定数量的操作指令来进行线程的强制切换；另一个方面是在线程释放 GIL 后，开始等待，直到某个其他线程获取 GIL 后，再开始尝试去获取 GIL，这样虽然可以避免此前获得 GIL 的线程，不会立即再次获取 GIL，但仍然无法保证优先级高的线程优先获取 GIL。这种方式只能解决部分问题，并未改变 GIL 的本质。

Python 提供了其他方式可以绕过 GIL 的局限，比如使用多进程 multiprocess 模块或者采用 C 语言扩展的方式，以及通过 ctypes 和 C 动态库来充分利用物理内核的计算能力。

建议 69：对象的管理与垃圾回收

通常来说 Python 并不需要用户自己来管理内存，它与 Perl、Ruby 等很多动态语言一样具备垃圾回收功能，可以自动管理内存的分配与回收。

Python 中内存管理的方式：Python 使用引用计数器（Reference counting）的方法来管理内存中的对象，即针对每一个对象维护一个引用计数值来表示该对象当前有多少个引用。当其他对象引用该对象时，其引用计数会增加 1，而删除一个队当前对象的引用，其引用计数会减 1。只有当引用计数的值为 0 时的时候该对象才会被垃圾收集器回收，因为它表示这个对象不再被其他对象引用，是个不可达对象。引用计数算法最明显的缺点是无法解决循环引用的问题，即两个对象相互引用。

循环引用常常会在列表、元组、字典、实例以及函数使用时出现。对于由循环引用而导致的内存泄漏的情况，可以使用 Python 自带的一个 gc 模块，它可以用来跟踪对象的“入引用（incoming reference）“和”出引用（outgoing reference）”，并找出复杂数据结构之间的循环引用，同时回收内存垃圾。有两种方式可以触发垃圾回收：一种是通过显式地调用 `gc.collect()` 进行垃圾回收；还有一种是在创建新的对象为其分配内存的时候，检查 threshold 阈值，当对象的数量超过 threshold 的时候便自动进行垃圾回收。默认情况下阈值设为（700，10，10），并且 gc 的自动回收功能是开启的，这些可以通过 `gc.isenabled()` 查看。

```
import gc
print(gc.isenabled())
print(gc.get_threshold())
```

一个解决循环引用内存回收的示例：

```
def main():
    collected = gc.collect()
    print("Garbage collector before running: collected {} objects.".format(collected))
    print("Creating reference cycles...")
    A = Leak()
    B = Leak()
    A.b = B
    B.a = A
    A = None
    B = None
    collected = gc.collect()
    print(gc.garbage)
    print("Garbage collector after running: collected {} objects".format(collected))

if __name__ == "__main__":
    ret = main()
    sys.exit(ret)
```

`gc.garbage` 返回的是由于循环引用而产生的不可达的垃圾对象的列表，输出为空表示内存中此时不存在垃圾对象。`gc.collect()` 显示所有收集和销毁的对象的数目，此处为 4（2 个对象 A、B，以及其实例属性 dict）。

如果在类 Leak 中添加析构方法 `__del__()`，会发现 `gc.garbage` 的输出不再为空，而是对象 A、B 的内存地址，也就是说这两个对象在内存中仍然以“垃圾”的形式存在。

实际上当存在循环引用并且当这个环中存在多个析构方法时，垃圾回收器不能确定对象析构的顺序，所以为了安全起见仍然保持这些对象不被销毁。而当环被打破时，gc 在回收对象的时候便会再次自动调用 `__del__()` 方法。

gc 模块同时支持 DEBUG 模式，当设置 DEBUG 模式之后，对于循环引用造成的内存泄漏，gc 并不释放内存，而是输出更为详细的诊断信息为发现内存泄漏提供便利，从而方便程序员进行修复。更多 gc 模块可以参考文档

第 7 章 使用工具辅助开发

Python 项目的开发过程，其实就是一个或多个包的开发过程，而这个开发过程又由包的安装、管理、测试和发布等多个节点构成，所以这是一个复杂的过程，使用工具进行辅助开发有利于减少流程损耗，提升生产力。比如 `setuptools`、`pip`、`paster`、`nose` 和 `Flask-PyPI-Proxy` 等。

建议 70：从 PyPI 安装包

PyPI 全称 Python Package Index，直译过来就是“Python 包索引”，它是 Python 编程语言的软件仓库，类似 Perl 的 CPAN 或 Ruby 的 Gems。可以通过包的名字查找、下载、安装 PyPI 上的包。对于包的作者，在 PyPI 上注册账号后，还可以登记、更新、上传包等。

> ### 注意
>
> PyPI 有良好的镜像机制，可以方便地在全球各地架设自有镜像，目前可用的几个镜像列出在 PyPI Mirrors 页面上，而各个镜像的同步情况可以在专门的网站上看到。豆瓣网架设了一个镜像，地址是 http://pypi.douban.com，访问速度很快，使用 pip 的 `--index` 参数。
>
> ### 注意
>
> 因为包在 PyPI 上的主页的 URL 都是 https://pypi.python.org/pypi/{package} 的形式，所以在知道包名的情况下，可以直接手动输入 URL

`setuptools` ，在 Ubuntu Linux 上，可以使用 apt 安装这个包：

```
sudo aptitude install python-setuptools
```

其他操作系统大同小异，运行其相应的包管理软件就可以安装。但如果使用 MS Windows，则需要去它的主页下载然后手动安装。

操作系统对应的软件仓库中的 `setuptools` 版本通常比较低，所以安装完成以后，最好执行以下命令将其更新到最新版本：

```
easy_install -U setuptools
```

`setuptools` 是来自 PEAK（Python Enterprise Application Kit，一个致力于提供 Python 开发企业级应用工具包的项目），由一组发布工具组成，方便程序员下载、构建、安装、升级和卸载 Python 包，它可以自动处理包的依赖关系。

> 注意
>
> 因为 PEAK 发展停滞，累及 setuptools 有好几年没有更新，所以有了一个分支项目，称为 distribute，很长一段时间里，运行 `easy_install -U setuptools` 更新安装的其实是 distribute。在 2013 年年初，distribute 合并到 setuptools，回归主分支了，而 distribute 项目也就不再维护了。

安装 `setuptools` 之后，就可以运行 `easy_install` 命令了。

建议 71：使用 pip 和 yolk 安装、管理包

`setuptools` 有几个缺点，比如功能缺失（不能查看已经安装的包、不能删除已经安装的包），也缺乏对 git、hg 等版本控制系统的原生支持。

pip 使用子命令形式的 CLI 接口。

【坑爹第 201 页缺失了， yolk 介绍没了】

下面介绍的是 pip 还不具备的功能：

`yolk --entry-map <package>` 可以显示包注册的所有入口点，这样可以了解到安装的包都提供了哪些命令行工具，或者支持哪些基于 `entry-point` 的插件系统。可以使用 `--entry-points` 参数查看哪些包实现了某一个包的插件协议。

如果使用的是桌面版的操作系统，利用 `yolk -H <package>` 可以打开一个浏览器，并将你指定的包显示在 PyPI 上的主页，从此告别手动拼接 URL 的历史。

建议 72：做 paster 创建包

个人编写的库，最好是像第三方库一样，可以方便地下载、安装、升级、卸载，也就是说能够放到 PyPI 上面，也能够很好地跟 pip 或者 yolk 这样的工具集成。Python 提供了这方面的支持，那即是 distutils 标准库，至少提供了以下几方面的内容：

- 支持包的构建、安装、发布（打包）
- 支持 PyPI 的登记、上传
- 定义了扩展命令的协议，包括 `distutils.cmd.Command` 基类、`distutils.commands` 和 `distutils.key_words` 等入口点，为 `setuptools` 和 `pip` 等提供了基础设施。

要使用 distutils，按习惯需要编写一个 `setup.py` 文件，作为后续操作的入口点。它的内容如下：

```
from distutils.core import setup
setup(name="arithmetic",
      version='1.0',
      py_modules=["your_script_name"],
      )
```

`setup.py` 文件的意义是执行时调用 `distutils.core.setup()` 函数，而实参是通过命名参数指定的。name 参数指定的是包名；version 指定的是版本；而 `py_modules` 参数是一个序列类型，里面包含需要安装的 Python 文件。

编写好 `setup.py` 文件以后，就可以使用 `python setup.py install` 进行安装了。

安装成功后可以使用 yolk 查看一下：

`yolk -l | grep your_script_name`

`distutils` 还带有其他命令，可以通过 `python setup.py --help-commands` 进行查询。

若要把包提交到 PyPI，还要遵循 PEP241，给出足够多的元数据才行，比如对包的简短描述、详细描述、作者、作者邮箱、主页和授权方式等。包含太多内容了，如果每一个项目都手写很困难，最好找一个工具可以自动创建项目的 `setup.py` 文件以及相关的配置、目录等。Python 中做这种事的工具有好几个，做得最好的是 `pastescript` 。`pastescript` 是一个有着良好插件机制的命令行工具，安装以后就可以使用 `paster` 命令，创建适用于 `setuptools` 的包文件结构。

安装好 `pastescript` 以后可以看到它注册了一个命令行入口 paster（还有许多插件协议的实现）：`yolk --entry-map pastescript`

接下来讨论一下 paster 怎么用

`paster --help` ，paster 支持多个命令，首先要学习的是 create。create 命令用于根据模板创建一个 Python 包项目，创建的项目文件结构可以使用 `setuptools` 直接构建，并且支持 `setuptools` 扩展的 `distutils` 命令，如 `develop` 命令等。

使用 `paster help create` 查看帮助，可以看到 `--template` 参数用以指定使用的模板，可以使用 `--list-templates` 查询一下目前安装的模板了。如果一个全新的环境只安装好了 `paster` ，那么一般只有两个模板：`basic_package` 和 `paste_deploy` 。

执行命令示例：`paster create -o arithmetic-2 -t basic_package arithmetic` ，可以看到，简单地填写几个问题以后，paster 就在 arithmetic-2 目录生成了名为 arithmetic 的包项目。可以看到 `setup.py` 文件中所有的参数都帮我们填好了。不过第一次使用 paster 的话，回答问题时可能会输入错误，而 paster 又不能回退删除输入，所以就只好重来一次。要解决这个问题，可以用上 `--config` 参数，它是一个类似 ini 文件格式的配置文件，可以在里面填好各个模板变量的值（查询模板有哪些变量用 `--list-variables` 参数），然后就可以使用了。示例如下：

```
[pastescript]
description = corp-prj
license_name =
keywords = Python
long_description = corp-prj
author = xxx corp
author_email = xxx@example.com
url = http://example.com
version = 0.0.1
```

然后这样使用：`paster create -t basic_package --config="corp-prj-setup.cfg" arithmetic`

建议 73：理解单元测试概念

单元测试用来验证程序单元的正确性，一般由开发人员完成，是测试过程的第一个环节，以确保缩写的代码符合软件需求和遵循开发目标。好的单元测试有以下好处：

- 减少了潜在 bug
- 大大缩减软件修复的成本
- 为集成测试提供基本保障

有效的单元测试应该从以下几个方面考虑：

- 测试先行，遵循单元测试步骤：
    - 创建测试计划（Test Plan）
    - 编写测试用例，准备测试数据
    - 编写测试脚本
    - 编写被测代码，在代码完成之后执行测试脚本
    - 修正代码缺陷，重新测试直到代码可接受为止
- 遵循单元测试基本原则：
    - 一致性
    - 原子性
    - 单一职责：测试应该基于情景（scenario）和行为，而不是方法。如果一个方法对应着多种行为，应该有多个测试用例；而一个行为即使对应多个方法也只能有一个测试用例
    - 隔离性：不能依赖于具体的环境设置，如数据库的访问、环境变量的设置、系统的时间等；也不能依赖于其他的测试用例以及测试执行的顺序，并且无条件逻辑依赖。单元测试的所有输入应该是确定的，方法的行为和结构应是可以预测的。
- 使用单元测试框架，在单元测试方面常见的测试框架有 PyUnit 等，它是 JUnit 的 Python 版本，在 Python2.1 之前需要单独安装，在 Python2.1 之后它成为了一个标准库，名为 unittest。它支持单元测试自动化，可以共享地进行测试环境的设置和清理，支持测试用例的聚集以及独立的测试报告框架。unittest 相关的概念主要有以下 4 个：
    - 测试固件（test fixtures）：测试相关的准备工作和清理工作，基于类 TestCase 创建

测试固件的时候通常需要重新实现 `setUp()` 和 `tearDown()` 方法。当定义了这些方法的时候，测试运行器会在运行测试之前和之后分别调用这两个方法
  - 测试用例（test case）：最小的测试单元，通常基于 TestCase 构建
  - 测试用例集（test suite）：测试用例的集合，使用 TestSuite 类来实现，除了可以包含 TestCase 外，也可以包含 TestSuite
  - 测试运行器（test runner）：控制和驱动整个单元测试过程，一般使用 TestRunner 类作为测试用例的基本执行环境，常用的运行器为 TextTestRunner，它是 TestRunner 的子类，以文字方式运行测试并报告结果。

用 TestSuite 类来组织 TestCase，TestSuite 类可以看成是 TestCase 类的一个容器，用来对多个测试用例进行组织，这样多个测试用例可以自动在一次测试中全部完成。

```
suite = unittest.TestSuite()
suite.addTest(MyCalTest("testAdd"))
suite.addTest(MyCalTest("testSub"))
runner = unittest.TextTestRunner()
runner.run(suite)
```

运行命令：`python -m unittest -v MyCalTest`

建议 74：为包编写单元测试

实际项目中的测试有不少麻烦：

- 程序员希望测试更加自动化
- 一个测试用例往往在测试之前需要进行打桩或做一些准备工作，在测试之后要清理现场，最好有一个框架可以自动完成这些工作
- 对于大项目，大量的测试用例需要分门分类地放置，而测试之后，分别产生相应的测试报告

unittest 框架，除了 自动匹配调用以 test 开头的方法之外，`unittest.TestCase` 还有模板方法 `setUp()` 和 `tearDown()`，通过覆盖这两个方法，能够实现在测试之前执行一些准备工作，并在测试之后清理现场。

可以使用 `unittest` 测试发现（test discover）功能：`python -m unittest discover`，unittest 将递归地查找当前目录下匹配 `test*.py` 模式的文件，并将其中 `unittest.TestCase` 的所有子类都实例化，然后调用相应的测试方法进行测试。

`unittest` 的测试发现功能是 Python2.7 版本中才有的，如果使用更旧的版本，需要安装 `unittest2`

除此之外，`setuptools` 对 `distutils.command` 进行了扩展，增加了命令 test。这个命令执行的时候，先运行 `egg_info` 和 `build_ext` 子命令构建项目，然后把项目路径加到 `sys.path` 中，再搜寻所有的测试套件（`test suite`，通常指多个测试用例或测试套件的组合），并运行之。要使用这个扩展命令，需要在调用 `setup()` 函数的时候向它传递 `test_suite` 元数据，例如：

```
# setup.py
...
setup(name = "arithmetic",
      ...
      test_suite = "test_arithmetic",
      ...
```

`test_suite` 元数据的值可以指向一个包、模块、类或函数，比如在 flask 项目中，是 `test_suite = "flask.testsuite.suite"`，其中 `flask.testsuite.suite` 是一个函数；而在 `arithmetic` 项目中，`test_arithmetic` 是一个模块。

使用 `setuptools` 的测试发现功能，可以给开发人员更一致的开发体验，就像使用 `build`、`install` 命令一样。但是来自 unittest 本身的缺陷让开发人员想要找到一个更好的测试框架：

- unittest 并不够 Pythonic，比如从 JUnit 中继承而来的首字母小写的骆驼命名法；所有的测试用例都需要从 TestCase 集成
- unittest 的 `setUp()` 和 `tearDown()` 只是在 TestCase 的层面上提供，即每一个测试用例执行的时候都会运行一遍，如果有多个模块需要测试，那么创建环境和清理现场操作都会带来大量工作
- unittest 没有插件机制进行功能扩展，比如想要增加测试覆盖统计特性就非常困难。

`nose` 就是作为更好的测试框架进入视线的，而它更是一个具有更强大的测试发现运行的程序。此外，nose 定义了插件机制，使得扩展 nose 的功能成为可能（默认自带 coverage 插件）。使用 pip 安装后，就多了一个 `nosetests` 命令可以使用：`nosetests -v`

可以看到 nose 能够自动发现测试用例，并调用执行，由于它与原有的 unittest 测试用例兼容，所以可以随时将它引入到项目中来。其实 nose 的测试发现机制更进一步，它抛弃了 unittest 中 测试用例必须放在 TestCase 子类中的限制，只要命名符合 `(?:^|[b_.-])[Tt]est` 正则表达式的类和函数都可作为测试用例运行。

此外，nose 作为一个测试框架，也提供了与 `unittest.TestCase` 类似的断言函数，但它抛弃了 `unittest` 的那种 Java 风格的命令方式，使用的是符合 PEP8 的命名方式。

针对 `unittest` 中 `setUp()` 和 `tearDown()` 只能放在 TestCase 中的问题，`nose` 提供了 3 个级别的解决方案，这些配置和清理函数，可以放在包（`__init__.py` 文件中）、模块和测试用例中，非常完全地解决了不同层次的测试需要的配置和清理需求。

最后，`nose` 与 `setuptools` 的集成更加友好，提供了 `nose.collector` 作为通过的测试套件，让开发人员无须针对不同项目编写不同的套件。需要对 `setup.py` 作如下修改：

```
# setup.py
setup(name = "arithmetic",
    ...
    # test_suite = "test_arithmetic",
    test_suite = "nose.collector",
    ...
```

然后运行 `python setup.py test`，得到的结果是一样的。因为使用 `nose.collector` 之后，`test_siuite` 元数据就确定不变了，所以它也非常适合写入 `paster` 的模板中去，在构建目录的时候自动生成。

建议 75：利用测试驱动开发提高代码的可测性

测试驱动开发（Test Driven Development，TDD）是敏捷开发中一个非常重要的理念，它提倡在真正开始编码之前测试先行，先编写测试代码，再在其基础上通过基本迭代完成编码，并不断完善。一般来说，遵循以下过程：

- 编写部分测试用例，并运行测试
- 如果测试通过，则回到测试用例编写的步骤，继续添加新的测试用例
- 如果测试失败，则修改代码直到通过测试
- 当所有测试用例编写完成并通过测试之后，再来考虑对代码进行重构

关于测试驱动开发和提高代码可测性方面有几点需要说明：

- TDD 只是手段而不是目的，因此在实践中尽量只验证正确的事情，并且每次仅仅验证一件事。当遇到问题时不要局限于 TDD 本身所涉及的一些概念，而应该回头想想采用 TDD 原本的出发点和目的是什么
- 测试驱动开发本身就是一门学问
- 代码的不可测性可以从以下几个方面考量：实践 TDD 困难；外部依赖太多；需要写很多模拟代码才能完成测试；职责太多导致功能模糊；内部状态过多且没有办法去操作和维护这些状态；函数没有明显返回或者参数过多；低内聚高耦合等等。

建议 76：使用 Pylint 检查代码风格

如果团队遵循 PEP8 编码风格，Pylint 是个不错的选择（还有其他选择，比如 pychecker、pep8 等）。Pylint 始于 2003 年，是一个代码分析工具，用于检查 Python 代码中的错误，查找不符合代码编码规范以及潜在的问题。支持不同的 OS 平台，如 Windows、Linux、OSX 等，特性如下：

- 代码风格审查。它以 Guido van Rossum 的 PEP8 为标准，能够检查代码的行长度，不

符合规范的变量名以及不恰当的模块导入等不符合编码规范的代码
- 代码错误检查。如未被实现的接口，方法缺少对应参数，访问模块中未定义的变量等
- 发现重复以及设计不合理的代码，帮助重构
- 高度的可配置化和可定制化，通过 pylintrc 文件的修改可以定义自己适合的规范
- 支持各种 IDE 和编辑器集成。如 Emacs、Eclipse、WingIDE、VIM、Spyder 等
- 能够基于 Python 代码生成 UML 图，Pylint0.15 中就集成了 Pyreverse，能够轻易生成 UML 图形
- 能够与 Hudson、Jenkins 等持续集成工具相结合支持自动代码审查

使用 Pylint 分析代码，输出分为两部分：一部分为源代码分析结果，第二部分为统计报告。报告部分主要是一些统计信息，总体来说有以下6 类：

- `Statistics by type` ：检查的模块、函数、类等数量，以及它们中存在文档注释以及不良命名的比例
- `Raw metrics` ：代码、注释、文档、空行等占模块代码量的百分比统计
- `Duplication` ：重复代码的统计百分比
- `Messages by category` ：按照消息类别分类统计的信息以及和上一次运行结果的对比
- `Messages` ：具体的消息 ID 以及它们出现的次数
- `Global evaluation` ：根据公式计算出的分数统计： `10.0 - ((float(5 * error + warning + refactor + convention) / statement) * 10)`

源代码分析主要以消息的形式显示代码中存在的问题，消息以 `MESSAGE_TYPE:LINE_NUM:[OBJECT:]MESSAGE` 的形式输出，主要分为以下 5 类：

- （C）惯例，违反了编码风格标准
- （R）重构，写得非常糟糕的代码
- （W）警告，某些 Python 特定的问题
- （E）错误，很可能是代码中的 bug
- （F）致命错误，阻止 Pylint 进一步运行的错误

如果信息输出 `trailing-whitespace` 信息，可以使用命令 `pylint --help-msg="trailing-whitespace"` 来查看，这个是行尾存在空格。

如果不希望对这类代码风格进行检查，可以使用命令行过滤掉这些类别的信息，比如 `pylint -d C0303,W0312 BalancePoint.py`

`Pylint` 支持可配置化，如果在项目中希望使用统一的代码规范而不是默认的风格来进行代码检查，可以指定 `--generate-rcfile` 来生成配置文件。默认的 `Pylintrc` 可以在 `Pylint` 的目录 examples 中找到。如默认支持的变量名的正则表达式为： `variable-rgx=[a-z_][a-z0-9_]{2,30}$` ，可以根据自己需要进行相应修改。其他配置如 reports 用于控制是否输出统计报告； `max-module-lines` 用于设置模块最大代码行数； `max-line-length` 用于设置代码行最大长度； `max-args` 用于设置函数的参数个数等。

建议 77：进行高效的代码审查

有效的代码审查流程非常必要，它能够使得大量 bug 在该阶段被发现，并且大幅度降低 bug 修复的总体代价，因为代码审查阶段 bug 的修复代价非常小。团队应该以这样的态度去看待代码审查：

- 不要错误地理解代码审查会的目的，代码审查会的首要目的是提高代码质量，找出 defect 或者设计上的不足而非修复 defect。
- 代码审查过程不应该有 KPI（Key Performance Indicator） 评价的成分
- 对管理层的建议：除非经理或者组长真正参与到具体的技术问题，否则应该尽量避免这些人员参与代码审查会，因为这些人直接与员工 KPI 挂钩
- 对开发者的建议：把代码审查当做一个学习的机会

定位角色

一般来说代码审查会上有 4 类角色：仲裁者、会议记录者、被评审开发人员和评审者。

- 仲裁者，一般由技术专家或者资深技术人员担任，主要职责是控制会议流程和时间，保证会议流程的高效性，同时能够在必要的时候给予评审技术指导
- 会议记录者：及时记录评审过程中发现的问题，包括问题的提出者、问题描述、问题产生的位置等
- 被评审开发人员：一般被评审开发人员在评审开始的时候应该对其代码有综合性的介绍，在评审者提出问题或者质疑的时候应该及时给出解释
- 评审者：除了以上 3 类人员外，其余与会的人都称为评审者

充分准备

在代码提交审查之前，代码作者需要对代码进行自我修正等

合理使用技术和工具

- 检查表（checklist）：检查表有利于有针对性地发现代码中存在的问题，如变量是否初始化，函数调用的参数，命名是否一致，字符串是否正确解码，有没有 import 未使用的 lib，逻辑操作符是否正确，（）、{} 对是否一致等
- 台面检查（Desk Checking）：适合在编码早期对顺序执行的代码进行检查，手工模拟代码的执行过程来检查程序中潜在的问题
- IRT（`Interleaving Review Technique`）：适合于并发性的代码或者容错性系统
- 代码审查工具，如 `Rietveld、review board、Collaborative Code Review Tool（CCRT）` 等

控制评审时间和评审内容

为了保证效率，一般来说一次评审时间要尽量控制在 45 分钟到 1 个小时。一次评审的代码行数应控制在 200 行以内，最好不超过 400 行。

关注技术层面，对事不对人

要把重点放在技术问题以及如何解决上，而不是诸如代码风格、时间进度之类的非技术层面的问题上。

记录问题，追踪进一步行动

不要忽视附加的培训作用

建议 78：将包发布到 PyPI

建立项目之后，添加了相应的业务代码，并通过测试之后，就可以考虑发布给下游用户了。如果是项目内部协作，把项目打一个 zip 包或者 tar ball 发出去，最简单不过了。另外 `setuptools` 仍然提供了完善的支持：

`sudo python setup.py sdist --formats=zip, gztar`

`setuptools` 的 `sdist` 命令的意思是构建一个源代码发行包，它将根据调用 `setup()` 函数时给定的实参将整个项目打包（和压缩）。根据当前的平台（操作系统）不同，产生的文件也是不一样的。一般在 MS Windows 系统下，产生 `.zip` 格式的压缩包，而在 GNU Linux 或 macOS 系统下，产生 `.tar.gz` 格式的压缩包。可以使用 `--formats` 参数，指定产生 `.zip` 格式和 `.tar.gz` 格式。最终产生的包文件放在 `./dist` 目录下。

产生这两个包以后，就可以发布给项目的下游合作者了。发布方式可以是邮件、FTP，或者直接使用 IM 传送。下游开发者收到后有两种安装方式：一种是解压缩，然后进入 `setup.py` 进行安装，另一种是使用 pip 安装。

如果是较大的团队一起协作一个项目，最好把包发布到 PyPI 上面。可以是 pypi.python.org 这个官方的 PyPI，也可以是团队架设的私有 PyPI。

标准库 distutils 自身已经带有发布到 PyPI 的功能，那就是 register 和 upload 命令。register 命令用以在 PyPI 上面注册一个包，这个包名必须是尚未使用的。在注册包名之前，先在 PyPI 上注册一个用户，可以通过 PyPI 网页注册，也可以直接使用 register 命令提供的选项注册。

验证通过以后，distutils 向 PyPI 申请注册包名，一般都能够成功。但如果这个包名已经被别的用户使用过了，那会引发一个 403 错误。

涉及到的命令如下：

```
python setup.py --help-commands
python setup.py register
python setup.py register -n
python setup.py sdist upload
```

第 8 章 性能剖析与优化

建议 79：了解代码优化的基本原则

代码优化是指在不改变程序运行结果的前提下使得程序运行的效率更高，优化的代码意味着代运行速度更快或者占用的资源更少。有以下几点原则：

- 优先保证代码是可工作的：让正确的程序更快要比让快速的程序正确容易得多。因此优化的前提是代码满足了基本的功能需求，是可工作的。过早地进行优化可能会忽视对总体性能指标的把握，忽略可移植性、可读性、内聚性等，更何况每个模块甚至每行优化的代码并不一定能够带来整体运行性能良好，因为性能瓶颈可能出现在意想不到的地方，比如模块与模块之间的交互和通信等。
- 权衡优化的代价：优化是有代价的，可能是面临着牺牲时间换空间或者空间换时间的选择；如果在项目时间紧迫的情况下能够仅仅通过增加硬件资源就解决主要性能问题，不妨选择更强大的部署环境；或者在已经实现的代码上进行修修补补试图进行优化代码所耗费的经历超过重构的代价时，重构可能是更好的选择
- 定义性能指标，集中力量解决首要问题：我们必须制定出可以衡量快的具体指标，比如在什么样的运行环境下（如网络速度、硬件资源等）、运行什么样的业务响应时间的范围是多少秒。
- 不要忽略可读性，优化不能以牺牲代码的可读性，甚至带来更多的副作用为代价。

建议 80：借助性能优化工具

好的工具能够对性能的提升起到非常关键的作用。常见的性能优化工具有 Psyco、Pypy 和 cPython 等。

- Psyco：Psyco 是一个 `just-in-time` 的编译器，它能够在不改变源代码的情况下提高一定的性能，Psyco 将操作编译成部分优化的机器码，其操作分成三个不同的级别，有“运行时”、“编译时”和“虚拟时”变量，并根据需要提高和降低变量的级别。运行时变量只是常规 Python 解释器处理的原始字节码和对象结构。一旦 Psyco 将操作编译成机器码，那么编译时变量就会在机器寄存器和可直接访问的内存位置中表示。同时 Python 能高速缓存已编译的机器码以备以后重用，这样能节省一点时间。但 Psyco 也有其缺点，其本身所占内存较大。2012 年 Psyco 项目停止维护并正式结束，由 Pypy 所接替。
- Pypy：Python 的动态编译器，是 Psyco 的后继项目。其目的是，做到 Psyco 没有做到的动态编译。Pypy 的实现分为两部分，第一部分“用 Python 实现的 Python”，实际上它是使用一个名为 RPython 的 Python 子集实现的，Pypy 能够将 Python 代码转成 C、.NET、Java 等语言和平台的代码；第二部分 Pypy 集成了一种编译 rPython 的即时（JIT）编译器，和许多编译器、解释器不同，这种编译器不关心 Python 代码的词法分析和语法树，所以它直接利用 Python 语言的 Code Object（Python 字节码的表示）。

Pypy 直接分析 Python 代码所对应的字节码，这些字节码既不是以字符形式也不是以某种二进制格式保存在文件中。

建议 81：利用 cProfile 定位性能瓶颈

程序运行慢的原因有很多，真正的原因往往是一两段设计并不那么良好的不起眼的程序，比如对一系列元素进行自定义的类型转换等。程序性能影响往往符合 80/20 法则，即 20% 的代码的运行时间占用了 80% 的总运行时间，我们需要一个工具帮忙定位性能瓶颈。

profile 是 Python 的标准库，可以统计程序里每一个函数的运行时间，并且提供了多样化的报表，而 cProfile 则是它的 C 实现版本，剖析过程本身需要消耗的资源更少。所以在 Python3 中，cProfile 代替了 profile，成为默认的性能剖析模块。使用 cProfile 来分析一个程序：

```
def foo():
    sum = 0
    for i in range(100):
        sum += i
    return sum
if __name__ == "__main__":
    import cProfile
    cProfile.run("foo()")
```

除了用这种方式，cProfile 还可以直接用 Python 解释器调用 cProfile 模块来剖析 Python 程序，如 `python -m cProfile prof1.py`

`cProfile` 的统计结果分为 `ncalls`、`tottime`、`percall`、`cumtime`、`percall`、`filename:lineno(function)` 等若干列。

统计项	意义
ncalls	函数的被调用次数
tottime	函数总计运行时间，不含调用的函数运行时间
percall	函数运行一次的平均时间，等于 tottime/ncalls
cumtime	函数总计运行时间，含调用的函数运行时间
percall	函数运行一次的平均时间，等于 cumtime/ncalls
filename:lineno(function)	函数所在的文件名、函数的行号、函数名

通常情况下，cProfile 的输出都直接输出到命令行，而且默认是按照文件名排序输出的。cProfile 简单地支持了一些需求，可以在 cProfile.run() 函数里再提供一个实参，就是保存输出的文件名。同样，在命令行参数里，也可以加多一个参数，用来保存 cProfile 的输出。

cProfile 解决了我们的对程序执行性能剖析的需求，但还有一个需求：以多种形式查看报表以便快速定位瓶颈。我们可以通过 pstats 模块的另一个类 Stats 来解决。Stats 的构造函数接受一个参数——就是 cProfile 的输出文件名。Status 提供了对 cProfile 输出结果进行排序、输出控制等功能。

```
if __name__ == "__main__":
    import cProfile
    cProfile.run("foo()", "prof.txt")
    import pstats
    p = pstats.Stats("prof.txt")
    p.sort_stats("time").print_stats()
```

Stats 有若干个函数，这些函数组合能输出不同的 cProfile 报表：

函数	函数的作用
`strip_dirs()`	用以除去文件名前面的路径信息
`add(filename,[...])`	把 profile 的输出文件加入 Stats 实例中统计
`dump_stats(filename)`	把 Stats 的统计结果保存到文件
`sort_stats(key, [...])`	把最重要的一个函数，用以排序 profile 的输出
`reverse_order()`	把 Stats 实例里的数据反序重排
`print_stats([restriction,...])`	把 Stats 报表输出到 stdout
`print_callers([restriction,...])`	输出调用了指定的函数的相关信息
`print_callees([restriction,...])`	输出指定的函数调用过的函数的相关信息

这里最重要的函数就是 `sort_stats` 和 `print_stats`，通过这两个函数我们几乎可以用适当的形式浏览所有的信息了：

- `sort_stats()` 接收一个或者多个字符串参数，如 time、name 等，表明要根据哪一列来排序。比如可以通过用 time 为 key 来排序得知最消耗时间的函数；也可以通过 cumtime 来排序，获知总消耗时间最多的函数。`sort_stats` 可接受的参数列表如下所示：
  - `ncalls`：被调用次数
  - `cumulative`：函数运行的总时间
  - `file`：文件名
  - `module`：模块名
  - `pcalls`：简单调用统计（兼容旧版，未统计递归调用）
  - `line`：行号
  - `name`：函数名
  - `nfl`：Name、file、line
  - `stdname`：标准函数名
  - `time`：函数内部运行时间（不计调用子函数的时间）

- `print_stats` 输出最后一次调用 `sort_stats` 之后得到的报表。`print_stats` 有多个可选参数，用以筛选输出的数据。`print_stats` 的参数可以是数字也可以是 Perl 风格的正则表达式。例子：`print_stats(".1", "foo:")`，这个语句表示将 stats 里的内容取前面 10%，然后再将包含 "foo:" 这个字符串的结果输出；`print_stats("foo:", ".1")`，这个语句表示将 stats 里的包含 "foo:" 字符串的内容的前 10% 输出；`print_stats(10)`，这个语句表示将 stats 里前 10 条数据输出。
    - 实际上，profile 输出结果的时候相当于如下调用了 Stats 的函数：`p.strip_dirs().sort_stats(-1).print_stats()`。其中，`sort_stats` 函数的参数是 -1，这是为了与旧版本兼容而保留的。`sort_stats` 可以接受 `-1、0、1、2` 之一，这 4 个数分贝对应 "stdname"、"calls"、"time" 和 "cumulative"。但如果你使用了数字为参数，那么 pstats 只按照第一个参数进行排序，其他参数将被忽略

除了编程接口外，pstats 还提供了友好的命令行交互环境，在命令行执行 `python -m pstats` 就可以进入交互环境，在交互环境里可以使用 read 或 add 指令读入或加载剖析结果文件，stats 指令用以查看报表，callees 和 callers 指令用以查看特定函数的被调用者和调用者。

测定向 list 里添加一个元素需要的时间，可以考虑使用 timeit 模块。

timeit 模块除了有非常友好的编程接口，也同样提供了友好的命令行接口。首先，timeit 模块包含了一个类 Timer，它的构造函数如下：

```
class Timer([stmt="pass"[, setup="pass"[, timer=<time function>]]])
```

stmt 参数是字符串形式的一个代码段，这个代码段将被评测运行时间：setup 参数用以设置 stmt 的运行环境；timer 可以由用户使用自定义精度的计时函数。

`timeit.Timer` 有 3 个成员函数：`timeit([number=1000000])`，`timeit()` 执行一次 Timer 构造函数中的 setup 语句之后，就重复执行 number 次 stmt 语句，然后返回总计运行消耗的时间；`repeat([repeat=3[, number=1000000]])`，`repeat()` 函数以 number 为参数调用 timeit 函数 repeat 次，并返回总计运行消耗的时间。`print_exc([file=None])`，`print_exec()` 函数以代替标准的 tracback，原因在于 `print_exec()` 会输出错行的源代码。

除了可以使用 timeit 的编程接口外，也可以在命令行里使用 timeit：

`python -m timeit [-n N] [-r N] [-s S] [-t] [-c] [-h] [statement ...]`，其中参数的定义如下：

- `-n N/--number=N`，statement 语句执行的次数
- `-r N/--repeat=N`，重复多少次调用 `timeit()`，默认为 3
- `-s S/--setup=S`，用以设置 `statement` 执行环境的语句，默认为 "pass"
- `-t/--time`，计时函数，除了 Windows 平台外默认使用 `time.time()` 函数
- `-c/--clock`，计时函数，Windows 平台默认使用 `time.clock()` 函数
- `-v/--verbose`，输出更大精度的计时数值
- `-h/--help`，简单的使用帮助

建议 82：使用 `memory_profiler` 和 `objgraph` 剖析内存使用

Python 还提供了一些工具可以用来查看内存的使用情况以及追踪内存泄漏（如 `memory_profiler`、`objgraph`、`cProfile`、`PySizer` 及 `Heapy` 等），或者可视化地显示对象之间的引用（如 `objgraph`），从而为发现内存问题提供更直接的证据。

- `memory_profiler`
    - 在 Windows 平台需要先安装依赖包 psutil。
    - `memory_profiler` 的使用非常简单，在需要进行内存分析的代码之前用 `@profile` 进行装饰，然后运行命令 `python -m memory_profiler 文件名`，便可以输出每一行代码的内存使用以及增长情况
- `Objgraph`
    - `Objgraph` 的功能大致可以分为以下三类：
        - 统计，如 `objgraph.count(typename[, objects])` 表示根据传入的参数显示被 gc 跟踪的对象的数目；`objgraph.show_most_common_types([limit=10, objects])` 表示显示常用类型对应的对象的数目
        - 定位和过滤对象，如 `objgraph.by_type(typename[, objects])` 表示根据传入的参数显示被 gc 跟踪的对象信息；`objgraph.at(addr)` 表示根据给定的地址返回对象
        - 遍历和显示对象图。如 `objgraph.show_refs(objs[, max_depth=3, extra_ignore=(), filter=None, too_many=10, highlight=None, filename=None, extra_info=None, refcounts=False])` 表示从对象 objs 开始显示对象引用关系图；`objgraph.show_backrefs(objs[, max_depth=3, extra_ignore=(), filter=None, too_many=10, highlight=None, filename=None, extra_info=None, refcounts=False])` 表示显示以 objs 的引用作为结束的对象关系图。

两个例子，一个是生成对象 x 的引用关系图：

```
>>> import objgraph
>>> x = ['a', '1', [2, 3]]
>>> objgraph.show_refs([x], filename="test.png")
```

显示常用类型不同类型对象的数目，限制输出前 3 行：

```
>>> objgraph.show_most_common_types(limit=3)
wrapper_descriptor            1031
function                      975
builtin_function_or_method    615
```

建议 83：努力降低算法复杂度

在当前的计算硬件资源发展形势下，对空间复杂度的关注远没有时间复杂度高，因此降低算法的复杂度主要集中在对其时间复杂度的考量。算法的时间复杂度是指算法需要消耗的时间资源，常使用大写字母 O 表示。复杂度大 O 的排序比较：

```
O(1) < O(log * n) < O(n) < O(n log n) < O(n^2) < O(c^n) < O(n!) < O(n^n)
```

需要特别说明，算法的复杂度分析的粒度非常重要，其前提一定是粒度相同的指令执行时间近似，不能将任意一行代码直接当做 O(1) 进行分析，比如调用函数。另外，算法复杂度分析建立在同一级别语言实现的基础上，如果 Python 代码中含有 C 实现的代码，千万不能混在一起进行评估。

Python 常见数据结构基本操作时间复杂度：

数据结构及操作	平均时间复杂度	最差时间复杂度
list 复制	O(n)	O(n)
list 追加、取元素的值，给某个元素赋值	O(1)	O(1)
list 插入、删除某个元素，迭代操作	O(n)	O(n)
list 切片操作	O(k)	O(k)
set x in s	O(1)	O(n)
set 并 s`	` t	O(len(s) + len(t))
set 交 s & t	O(min(len(s), len(t)))	O(len(s) * len(t))
set 差 s - t	O(len(s))	
dict 获取修改元素的值，删除	O(1)	O(n)
dict 迭代操作	O(n)	O(n)
collections.deque 入列、出列（包括左边出入列）	O(1)	O(1)
collections.deque 扩大队列	O(k)	O(k)
collections.deque 删除元素	O(n)	O(n)

建议 84：掌握循环优化的基本技巧

循环的优化应遵循的原则是尽量减少循环过程中的计算量，多重循环的情形下尽量将内层的计算提到上一层。

- 减少循环内部的计算
- 将显式循环改为隐式循环，如果用 for 循环求解 1...n 的和，对比用公式求解。当然这可能会带来另一个负面影响：牺牲了代码的可读性。因此这种情况下清晰、恰当的注释是

非常必要的
- 在循环中尽量引用局部变量，在命名空间中局部变量优先搜索，因此局部变量的查询会比全局变量要快，当在循环中需要多次引用某一个变量的时候，尽量将其转换为局部变量，比如下面第二个循环就优于第一个循环：

```
# 较慢
x = [10, 34, 56, 78]
def f(x):
    for i in range(len(x)):
        x[i] = math.sin(x[i])
    return x

# 较快
def g(x):
    loc_sin = math.sin
    for i in range(len(x)):
        x[i] = loc_sin(x[i])
    return x
```

- 关注内层嵌套循环，尽量将内层循环的计算往上层移，比如：

```
# 较慢
for i in range(len(v1)):
    for j in range(len(v2)):
        x = v1[i] + v2[j]

# 较快
for i in range(len(v1)):
    v1i = v1[i]
    for j in range(len(v2)):
        x = v1i + v2[j]
```

建议 85：使用生成器提高效率

生成器的概念是，如果一个函数体中包含有 yield 语句，则称为生成器（generator），它是一种特殊的迭代器（iterator），也可以称为可迭代对象（iterable）。

实际上当需要在循环过程中依次处理一个序列中的元素的时候，就应该考虑生成器。yield 语句与 return 语句相似，当解释器执行遇到 yield 的时候，函数会自动返回 yield 语句之后的表达式的值。不过与 return 不同的是，yield 语句在返回的同时会保存所有的局部变量以及现场信息，以便在迭代器调用 `next()` 或 `send()` 方法的时候还原，而不是直接交给垃圾回收器（`return()` 方法返回后这些信息会被垃圾回收器处理）。这样就能够保证对生成器的每一次迭代都会返回一个元素，而不是一次性在内存中生成所有的元素。自 Python2.5 开始，yield 语句变为表达式，可以直接将其值赋给其他变量。

生成器的优点总体来说有如下几条：

- 生成器提供了一种更为便利的产生迭代器的方式，用户一般不需要自己实现 `__iter__` 和 next 方法，它默认返回一个迭代器
- 代码更为简洁优雅
- 充分利用了延迟评估（Lazy evaluation）的特性，仅在需要的时候才产生对应的元素，而不是一次生成所有的元素，从而节省了内存空间，提高了效率。
- 使得协同程序更为容易实现。协同程序是有多个进入点，可以挂起恢复的函数，这基本就是 yield 的工作方式。Python2.5 之后生成器的功能更完善，加入了 `send()` 、`close()` 和 `throw()` 方法。其中 `send()` 不仅可以传递值给 yield 语句，而且能够恢复生成器，因此生成器能大大简化协同程序的实现。

建议 86：使用不同的数据结构优化性能

考虑到 Python 中的查找、排序常用算法都已经优化到了极点（虽然对 `sort()` 使用 key 参数比使用 cmp 参数有更高的性能仍然值得一提），首先应当想到的是使用不同的数据结构优化性能。

`list` ，它的内存管理类似 C++ 的 `std::vector` ，即预先分配一定数量的内存，用完时，又继续往里面插入元素，会启动新一轮的内存分配。`list` 对象会根据内存增长算法申请一块更大的内存，然后将原有的所有元素拷贝过去，销毁之前的内存，再插入新元素。当删除元素时，也是类似，删除后发现已用空间比预分配空间的一半还少时，list 会另外申请一块小内存，再做一次元素拷贝，然后销毁原有的大内存。可见，如果 list 对象经常有元素数量的巨变，应当考虑使用 deque。

deque 就是双端队列，同时具备栈和队列的特性，能够提供在两端插入和删除时复杂度为 O(1) 的操作。相对于 list，它最大的优势在于内存管理方面。如果不熟悉 C++ 的 `std::deque` ，可以把 deque 想象为多个 list 连在一起，它的每一个 list 也可以存储多个元素。它的优势在于插入时，已有空间已经用完，那么它会申请一个新的内存空间来容纳新的元素，并将其与已有的其他内存空间串接起来，从而避免元素拷贝；在删除元素时也类似，无需移动元素。所以当元素数量巨变时，它的性能比 list 要好上许多倍。

对于 list 这种序列容器来说，除了 `pop(0)` 和 `insert(0, v)` 这种插入操作非常耗时之外，查找一元素是否在其中，也是 O(n) 的线性复杂度。在 C 语言中，标准库函数 `bsearch()` 能够通过二分查找算法在有序队列中快速查找是否存在某一元素。在 Python 中，对保持 list 对象有序以及在有序队列中查找元素有非常好的支持，这是通过标准库 bisect 来实现的。

`bisect` 并没有实现一种新的“数据结构”，其实它是用来维护“有序列表”的一组函数，可以兼容所有能够随机存取的序列容器，比如 list。它可使在有序列表中查找某一元素变得非常简单。

```
def index(a, x):
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError
```

保持列表有序需要付出额外的维护工作，但如果业务需要在元素较多的列表中频繁查找某些元素是否存在或者需要频繁地有序访问这些元素，使用 bisect 则相当值得。

对于序列容器，除了插入、删除、查找之外，还有一种很常见的需求是获取其中的极大值或极小值元素，这时候，可以使用 heapq 模块。类似 bisect，heapq 也是维护列表的一组函数，其中 `heapify()` 的作用是把一个序列容器转化为一个堆。

```
>>> import heapq
>>> import random
>>> alist = [random.randint(0, 100) for i in range(10)]
>>> heapq.heapify(alist)
```

可以看到，转化为堆后，alist 的第一个元素 `alist[0]` 是整个列表中最小的元素，heapq 将保证这一点，从而保证从列表中获取最小值元素的时间复杂度是 O(1)

除了通过 `heapify()` 函数将一个列表转换为堆之外，也可以通过 `heappush()`、`heappop()` 函数插入、删除元素，针对常见的先插入新元素再获取最小元素、先获取最小元素再插入新元素的需求，还有 `heappushpop(heap, item)` 和 `heapreplace(heap, item)` 函数可以快速完成。另外可以看出，每次元素增减之后的序列变化很大，所以千万不要乱用 heapq，以免带来性能问题。

另外，heapq 还有 3 个通用函数值得介绍，其中 `merge()` 能够把多个有序列表归并为一个有序列表（返回迭代器，不占用内存），而 `nlargest()` 和 `nsmallest()` 类似于 C++ 中的 `std::nth_element()`，能够返回无序列表中最大或最小的 n 个元素，并且性能比 `sorted(iterable, key=key)[:n]` 要高。

除了对容器的操作可能会出现性能问题外，容器中存储的元素也有很大的优化空间，在很多业务中，容器存储的元素往往是同一类型的，比如都是整数，此时就可以用 array 优化程序性能：

```
>>> import array
>>> a = array.array("c", "i_am_a_string") # 'c'表示存储的每个元素都相当于 C 语言中的 char 类型，占用内存大小为 1 字节
```

array 对象与 str 不同，它是可变对象，可以随意修改某一元素的值。

从容器到字符串的转变可以看出 array 的性能提升是比较大的，但也不能认为 array 在什么方面都有更好的性能，比如使用 `reverse()` 方法时。所以性能优化一定要根据 profiler 的剖析结果来进行。

建议 87：充分利用 set 的优势

Python 中集合是通过 Hash 算法实现的无序不重复的元素集。

集合中常见的操作及其对应的时间复杂度如下：

- `s.union(t)`：
    - s 和 t 的并集，平均时间复杂度为：`O(len(s) + len(t))`
- `s.intersection(t)`：
    - s 和 t 的交集，平均时间复杂度为：`O(min(left(s) + len(t)))`，最差为：`O(len(s) * len(t))`
- `s.difference(t)`：
    - s 和 t 的差集，s-t，在 s 中存在但在 t 中不存在的元素组成的集合，平均时间复杂度为：`O(len(s))`
- `s.symmetric_difference(t)`：
    - s ^t，s 和 t 的并集减去 s 和 t 的交集，平均时间复杂度：`O(len(s))`，最差时间复杂度：`O(len(s) * len(t))`

基本操作的复杂度基本为 O(n)，最差的情况下时间复杂度才为 O(n^2)。从测试数据中可以看出，实际上 set 的 union、intersection、difference 等操作要比 list 的迭代要快。因此如果涉及求 list 交集、并集或者差等问题可以转换为 set 来操作。

建议 88：使用 `multiprocess` 克服 GIL 的缺陷

GIL 的存在使得 Python 中的多线程无法充分利用多核的优势来提高性能。多进程 `Multiprocess` 是 Python 中的多进程管理包，在 Python2.6 版本中引进的，主要用来帮助处理进程的创建以及它们之间的通信和相互协调。它主要解决了两个问题：一是尽量缩小平台之间的差异，提供高层次的 API 从而使得使用者忽略底层 IPC 的问题；二是提供对复杂对象的共享支持，支持本地和远程并发。

类 Process 是 multiprocess 中较为重要的一个类，用户创建进程，其构造函数如下：`Process([group[, target[, name[, args[, kwargs]]]]])`

其中，参数 target 表示可调用对象；args 表示调用对象的位置参数元组；kwargs 表示调用对象的字典；name 为进程的名称；group 一般设置为 None。该类提供的方法与属性基本上与 threading.Thread 一致，包括 `is_alive()、join([timeout])、run()、start()、terminate()、daemon（要通过 start() 设置）、exitcode、name、pid` 等。

不同于线程，每个进程都有其独立的地址空间，进程间的数据空间也相互独立，因此进程之间数据的共享和传递不如线程来得方便。庆幸的是 `multiprocess` 模块中都提供了相应的机制：如进程间同步操作原语 Lock、Event、Condition、Semaphore，传统的管道通信机制 pipe 以及队列 Queue，用于共享资源的 `multiprocess.Value` 和 `multiprocess.Array` 以及 `Manager` 等。

`Multiprocessing` 模块在使用上需要注意以下几个要点：

- 进程之间的的通信优先考虑 Pipe 和 Queue，而不是 Lock、Event、Condition、Semaphore 等同步原语。进程中的类 Queue 使用 pipe 和一些 locks、semaphores 原语来实现，是进程安全的。该类的构造函数返回一个进程的共享队列，其支持的方法和线程中的 Queue 基本类似，除了方法 `task_done()` 和 `join()` 是在其子类 `JoinableQueue` 中实现的以外。需要注意的是，由于底层使用 pipe 来实现，使用 Queue 进行进程之间的通信的时候，传输的对象必须是可以序列化的，否则 put 操作会导致 PicklingError。此外，为了提供 put 方法的超时控制，Queue 并不是直接将对象写到管道中而是先写到一个本地的缓存中，再将其从缓存中放入 pipe 中，内部有个专门的线程 feeder 负责这项工作。由于 feeder 的存在，Queue 还提供了以下特殊方法来处理进程退出时缓存中仍然存在数据的问题。
    - `close()`：表明不再存放数据到 queue 中。一旦所有缓冲的数据刷新到管道，后台线程将退出。
    - `join_thread()`：一般在 close 方法之后使用，它会阻止直到后台线程退出，确保所有缓冲区中的数据已经刷新到管道中。
    - `cancel_join_thread()`：需要立即退出当前进程，而无需等待排队的数据刷新到底层管道的时候可以使用该方法，表明无须阻止到后台进程的退出。

`Multiprocessing` 中还有个 SimpleQueue 队列，它是实现了锁机制的 pipe，内部去掉了 buffer，但没有提供 put 和 get 的超时处理，两个动作都是阻塞的。

除了 `multiprocessing.Queue` 之外，另一种很重要的通信方式是 `multiprocessing.Pipe`。它的构造函数为 `multiprocess.Pipe([duplex])`，其中 `duplex` 默认为 True，表示为双向管道，否则为单向。它返回一个 Connection 对象的组（conn1, conn2），分别表示管道的两端。Pipe 不支持进程安全，因此当有多个进程同时对管道的一端进行读操作或者写操作的时候可能会导致数据丢失或者损坏。因此在进程通信的时候，如果是超过 2 个以上的线程，可以使用 queue，但对于两个进程之间的通信而言 Pipe 性能更快。

```
from multiprocessing import Process, Pipe, Queue
import time

def reader_pipe(pipe):
    output_p, input_p = pipe    # 返回管道的两端
    inout_p.close()
    while True:
        try:
            msg = output_p.recv()    # 从 pipe 中读取消息
```

```
        except EOFError:
                break

def writer_pipe(count, input_p):    # 写消息到管道中
    for i in range(0, count):
        input_p.send(i)                # 发送消息

def reader_queue(queue):            # 利用队列来发送消息
    while True:
        msg = queue.get()            # 从队列中获取元素
        if msg == "DONE":
            break

def writer_queue(count, queue):
    for ii in range(0, count):
        queue.put(ii)                # 放入消息队列中
    queue.put("DONE")

if __name__ == "__main__":
    print("testing for pipe:")
    for count in [10 ** 3, 10 ** 4, 10 ** 5]:
        output_p, input_p = Pipe()
        reader_p = Process(target=reader_pipe, args=((output_p, input_p),))
        reader_p.start()            # 启动进程
        output_p.close()
        _start = time.time()
        writer_pipe(count, input_p)    # 写消息到管道中
        input_p.close()
        reader_p.join()                # 等待进程处理完毕
        print("Sending {} numbers to Pipe() took {} seconds".format(count, (time.time(
) - _start)))

    print("testsing for queue:")
    for count in [10 ** 3, 10 ** 4, 10 ** 5]:
        queue = Queue()                # 利用 queue 进行通信
        reader_p = Process(target=reader_queue, args=((queue),))
        reader_p.daemon = True
        reader_p.start()

        _start = time.time()
        writer_queue(count, queue)    # 写消息到 queue 中
        reader_p.join()
        print("Seding {} numbers to Queue() took {} seconds".format(count, (time.time(
) - _start)))
```

从函数输出可以看出，pipe 所消耗的时间较小，性能更好。

- 尽量避免资源共享。相比于线程，进程之间资源共享的开销较大，因此要尽量避免资源共享。但如果不可避免，可以通过 `multiprocessing.Value` 和 `multiprocessing.Array` 或者 `multiprocessing.sharedctpyes` 来实现内存共享，也可以通过服务器进程管理器 Manager() 来实现数据和状态的共享。这两种方式各有优势，总体来说共享内存的方式更

快，效率更高，但服务器进程管理器 Manager() 使用起来更为方便，并且支持本地和远程内存共享。

使用 Value 进行内存共享

```
import time
from multiprocessing import Process, Value

def func(val):     # 多个进程同时修改 val
    for i in range(10):
        time.sleep(0.1)
        val.value += 1

if __name__ == "__main__":
    v = Value("i", 0)     # 使用 value 来共享内存
    processList = [Process(target=func, args=(v,)) for i in range(10)]
    for p in processList:
        p.start()
    for p in processList:
        p.join()
    print v.value
```

Python 官方文档中有个容易让人迷惑的描述：在 Value 的构造函数 `multiprocessing.Value(typecode_or_type, *args[, lock])` 中，如果 `lock` 的值为 True 会创建一个锁对象用于同步访问控制，该值默认为 True。因此很多人会以为 Value 是进程安全的，实际上真正要控制同步访问，需要实现获取这个锁：

```
def func(val):
    for i in range(10):
        time.sleep(0.1)
        with val.get_lock():     # 仍然需要使用 get_lock 方法来获取锁对象
            val.value += 1
```

使用 Manager 进行内存共享

```
import multiprocessing
def f(ns):
    ns.x.append(1)
    ns.y.append("a")

if __name__ == "__main__":
    manager = multiprocessing.Manager()
    ns = manager.Namespace()
    ns.x = []     # manager 内部包括可变对象
    ns.y = []

    print("before process operation: {}".format(ns))
    p = multiprocessing.Process(target=f, args=(ns,))
    p.start()
    p.join()
    print("after process operation {}".format(ns))    # 修改根本不会生效
```

`manager` 对象仅能传播对一个可变对象本身所做的修改，如有一个 `manager.list()` 对象，管理列表本身的任何更改会传播到所有其他进程。但是，如果容器对象内部还包括可修改的对象，则内部可修改对象的任何更改都不会传播到其他进程，因此，正确的处理方式：

```
import multiprocessing
def f(ns, x, y):
    x.append(1)
    y.append("a")
    ns.x = x     # 将可变对象也作为参数传入
    ns.y = y

if __name__ == "__main__":
    manager = multiprocessing.Manager()
    ns = manager.Namespace()
    ns.x = []     # manager 内部包括可变对象
    ns.y = []

    print("before process operation: {}".format(ns))
    p = multiprocessing.Process(target=f, args=(ns, ns.x, ns.y))
    p.start()
    p.join()
    print("after process operation {}".format(ns))
```

- 注意平台之间的差异。由于 Linux 平台使用 `fork()` 来创建进程，因此父进程中所有的资源，如数据结构、打开的文件或者数据库的连接都会在子进程中共享，而 Windows 平台中父子进程相对独立，因此为了更好地保持平台的兼容性，最好能够将相关资源对象作为子进程的构造函数的参数传递进去。要避免如下方式：

```
f = None
def child(f):
    # do something

if __name__ == "__main__":
    f = open(filename, mode)
    p = Process(target=child)
    p.start()
    p.join()
```

而推荐使用如下方式：

```
def child(f):
    print(f)

if __name__ == "__main__":
    f = open(filename, mode)
    p = Process(target=child, args=(f, ))    # 将资源对象作为构造函数参数传入
    p.start()
    p.join()
```

需要注意的是，Linux 平台上 multiprocessing 的实现是基于 C 库中的 fork()，所有子进程与父进程的数据是完全相同，因此父进程中所有的资源，如数据结构、打开的文件或者数据库的连接都会在子进程中共享。但 Windows 平台上由于没有 `fork()` 函数，父子进程相对独立，因此保持了平台的兼容性，最好在脚本中加上 `if __name__ == "__main__"` 的判断，这样可以避免出现 RuntimeError 或者死锁。

- 尽量避免使用 `terminate()` 方式终止进程，并且确保 `pool.map` 中传入的参数是可以序列化的。

解决序列化问题，一个可行的正确做法如下：

```
import multiprocessing
def unwrap_self_f(*args, **kwargs):
    return calculate.f(*args, **kwargs)    # 返回一个对象

class calculate(object):
    def f(self, x):
        return x * x
    def run(self):
        p = multiprocessing.Pool()
        return p.map(unwrap_self_f, zip([self] * 3, [1, 2, 3]))

if __name__ == "__main__":
    c1 = calculate()
    print(c1.run())
```

建议 89：使用线程池提高效率

我们知道线程的生命周期分为 5 个状态：创建、就绪、运行、阻塞和终止。自线程创建到终止，线程便不断在运行、就绪和阻塞这 3 个状态之间转换直至销毁。而真正占有 CPU 的只有运行、创建和销毁这 3 个状态。一个线程的运行时间由此可以分为 3 部分：线程的启动时间（Ts）、线程体的运行时间（Tr）以及线程的销毁时间（Td）。在多线程处理的情境中，如果线程不能够被重用，就意味着每次创建都需要经过启动、销毁和运行这 3 个过程。这必然会增加系统的相应时间，降低效率。而线程体的运行时间 Tr 不可控制，在这种情况下要提高线程运行的效率，线程池便是一个解决方案。

线程池通过实现创建多个能够执行任务的线程放入池中，所要执行的任务通常被安排在队列中。通常情况下，需要处理的任务比线程的数目要多，线程执行完当前任务后，会从队列中取下一个任务，直到所有的任务已经完成。

由于线程预先被创建并放入线程池中，同时处理完当前任务之后并不销毁而是被安排处理下一个任务，因此能够避免多次创建线程，从而节省线程创建和销毁的开销，带来更好的性能和系统稳定性。线程池技术适合处理突发性大量请求或者需要大量线程来完成任务、但任务实际处理时间较短的应用场景，它能有效避免由于系统中创建线程过多而导致的系统性能负载过大、响应过慢等问题。

Python 中利用线程池有两种解决方案：一是自己实现线程池模式，二是使用线程池模块。

一个线程池模式的简单实现

```
import Queue, sys, threading
import urllib2, os

# 处理 request 的工作线程
class Worker(threading.Thread):
    def __init__(self, workQueue, resultQueue, **kwargs):
        threading.Thread.__init__(self, **kwargs)
        self.setDaemon(True)
        self.workQueue = workQueue
        self.resultQueue = resultQueue

    def run(self):
        while True:
            try:
                callable, args, kwargs = self.workQueue.get(False)    # 从队列中取出一个
任务
                res = callable(*args, **kwargs)
                self.resultQueue.put(res)    # 存放处理结果到队列中
            except Queue.Empty:
                break

class WorkManager:    # 线程池管理器
    def __init__(self, num_of_workers=10):
```

```
        self.workQueue = Queue.Queue()    # 请求队列
        self.resultQueue = Queue.Queue()    # 输出结果的队列
        self.workers = []
        self._recruitThreads(num_of_workers)

    def _recruitThreads(self, num_of_workers):
        for i in range(num_of_workers):
            worker = Worker(self.workQueue, self.resultQueue)    # 创建工作线程
            self.workers.append(worker)    # 加入线程队列中

    def start(self):    # 启动线程
        for w in self.workers:
            w.start()

    def wait_for_complete(self):
        while len(self.workers):
            worker = self.workers.pop()    # 从池中取出一个线程处理请求
            worker.join()
            if worker.isAlive() and not self.workQueue.empty():
                self.workers.append(worker)    # 重新加入线程池中
        print("All jobs were completed")

    def add_job(self, callable, *args, **kwargs):
        self.workQueue.put((callable, args, kwargs))    # 在工作队列中加入青丘

    def get_result(self, *args, **kwargs):    # 获取处理结果
        return self.resultQueue.get(*args, **kwargs)

    def download_file(url):
        print("begin download {}".format(url))
        urlhandler = urllib2.urlopen(url)
        fname = os.path.basename(url) + ".html"
        with open(fname, "wb") as f:
            while True:
                chunk = urlhandler.read(1024)
                if not chunk:
                    break
                f.write(chunk)

urls = ["http://wiki.python.org/moni/WebProgramming",
       "https://www.createspace.com/3611970",
       "http://wiki.python.org/moin/Documention"]
wm = WorkerManager(2)    # 创建线程池
for i in urls:
    wm.add_job(download_file, i)    # 将所有请求加入队列中
wm.start()
wm.wait_for_complete()
```

自行实现线程池，需要定义一个 Worker 处理工作请求，定义 WorkerManager 来进行线程池的管理和创建，它包含一个工作请求队列和执行结果队列，具体的下载工作通过 `download_file()` 方法来实现。

相比自己实现的线程池模型，使用现成的线程池模块往往更简单。Python 中线程池模块的下载地址为：https://pypi.python.org/pypi/threadpool 。该模块提供了以下基本类和方法：

- `threadpool.ThreadPool` ：线程池类，主要的作用是用来分派任务请求和收集运行结果。主要有以下方法：
  - `__init__(self, num_workers, q_size=0, resq_size=0, poll_timeout=5)` ：建立线程池，并启动对应 `num_workers` 的线程； `q_size` 表示任务请求队列的大小， `resq_size` 表示存放运行结果队列的大小。
  - `createWorkers(self, num_workers, poll_timeout=5)` ：将 `num_workers` 数量对应的线程加入线程池中。
  - `dismissWorkers(self, num_workers, do_join=False)` ：告诉 `num_workers` 数量的工作线程当执行完当前任务后退出
  - `joinAllDismissedWorkers(self)` ：在设置为退出的线程上执行 `Thread.join`
  - `putRequest(self, request, block=True, timeout=None)` ：将工作请求放入队列中
  - `poll(self, block=False)` ：处理任务队列中新的请求
  - `wait(self)` ：阻塞用于等待所有执行结果。注意当所有执行结果返回后，线程池内部的线程并没有销毁，而是在等待新的任务。因此， `wait()` 之后仍然可以再次调用 `pool.putRequests()` 往其中添加任务
- `threadpool.WorkRequest` ：包含有具体执行方法的工作请求类
- `threadpool.WorkerThread` ：处理任务的工作线程，主要有 `run()` 方法以及 `dismiss()` 方法。
- `makeRequests(callable_, args_list, callback=None, exec_callback=_handle_thread_exception)` ：主要函数，作用是创建具有相同的执行函数但参数不同的一系列工作请求。

用线程池实现的例子

```
import urllib2
import os
import time
import threadpool

def download_file(url):
    print("begin download {}".format(url ))
    urlhandler = urllib2.urlopen(url)
    fname = os.path.basename(url) + ".html"
    with open(fname, "wb") as f:
        while True:
            chunk = urlhandler.read(1024)
            if not chunk:
                break
            f.write(chunk)

urls = ["http://wiki.python.org/moni/WebProgramming",
       "https://www.createspace.com/3611970",
       "http://wiki.python.org/moin/Documention"]
pool_size = 2
pool = threadpool.ThreadPool(pool_size)    # 创建线程池，大小为 2
requests = threadpool.makrRequests(download_file, urls)    # 创建工作请求
[pool.putRequest(req) for req in requests]

print("putting request to pool")
pool.putRequest(threadpool.WorkRequest(download_file, args=["http://chrisarndt.de/projects/threadpool/api/",]))    # 将具体的请求放入线程池
pool.putRequest(threadpool.WorkRequest(download_file, args=["https://pypi.python.org/pypi/threadpool",]))
pool.poll()    # 处理任务队列中的新的请求
pool.wait()
print("destory all threads before exist")
pool.dismissWorkers(pool_size, do_join=True)    # 完成后退出
```

建议 90：使用 C/C++ 模块扩展高性能

Python 具有良好的可扩展性，利用 Python 提供的 API，如宏、类型、函数等，可以让 Python 方便地进行 C/C++ 扩展，从而获得较优的执行性能。所有这些 API 却包含在 Python.h 的头文件中，在编写 C 代码的时候引入该头文件即可。

- 先用 C 实现相关函数，也可以直接使用 C 语言实现相关函数功能后再使用 Python 进行包装。

```
#include "Python.h"
static PyObject * pr_isprime(PyObject, *self, PyObject * args) {
  int n, num;
  if (!PyArg_ParseTuple(args, "i", &num))    // 解析参数
    return NULL;
  if (num < 1) {
    return Py_BuildValue("i", 0);    // C 类型的数据结构转换成 Python 对象
  }
  n = num - 1;
  while (n > 1) {
    if (num % n == 0) {
      return Py_BuildValue("i", 0);
      n--;
    }
  }
  return Py_BuildValue("i", 1);
}

static PyMethodDef PrMethods[] = {
  {"isPrime", pr_isprime, METH_VARARGS, "check if an input number is prime or not."},
  {NULL, NULL, 0, NULL}
};

void initpr(void) {
  (void) Py_InitModule("pr", PrMethods);
}
```

上面的代码包含以下 3 部分：

- 导出函数：C 模块对外暴露的接口函数 `pr_isprime`，带有 self 和 args 两个参数，其中参数 args 中包含了 Python 解释器要传递给 C 函数的所有参数，通常使用函数 `PyArg_ParseTuple()` 来获得这些参数值
- 初始化函数：以便 Python 解释器能够对模块进行正确的初始化，初始化时要以 init 开头，如 `initp`
- 方法列表：提供给外部的 Python 程序使用的一个 C 模块函数名称映射表 `PrMethods`。它是一个 PyMethodDef 结构体，其中成员依次表示方法名、导出函数、参数传递方式和方法描述。

```
struct PyMethodDef {
  char * m1_name;         // 方法名
  PyCFunction m1_meth;    // 导出函数
  int m1_flags;           // 参数传递方法
  char * m1_doc;          // 方法描述
}
```

参数传递方法一般设置为 `METH_VARARGS`，如果想传入关键字参数，则可以将其与 `METH_KEYWORDS` 进行或运算。若不想接受任何参数，则可以将其设置为 `METH_NOARGS`。该结构体必须与 `{NULL, NULL, 0, NULL}` 所表示的一条空记录来结尾。

- 编写 `setup.py` 脚本

```
from distutils.core import setup, Extension
module = Extension("pr", sources=["testextend.c"])
setup(name="Pr test", version="1.0", ext_modules=[module])
```

- 使用 `python setup.py build` 进行编译，系统会在当前目录下生成一个 `build` 子目录，里面包含 `pr.so` 和 `pr.o` 文件。
- 将生成的文件 `py.so` 复制到 Python 的 `site_packages` 目录下，或者将 `pr.so` 所在目录的路径添加到 `sys.path` 中，就可以使用 C 扩展的模块了。

更多关于 C 模块扩展的内容可以参考

建议 91：使用 Cython 编写扩展模块

`Python-API` 让大家可以方便地使用 `C/C++` 编写扩展模块，从而通过重写应用中的瓶颈代码获得性能提升。但是，这种方式仍然有几个问题：

- 掌握 C/C++ 编程语言、工具链有巨大的学习成本
- 即便是 C/C++ 熟手，重写代码也有非常多的工作，比如编写特定数据结构、算法的 C/C++ 版本，费时费力还容易出错

所以整个 Python 社区都在努力实现一个 "编译器“，它可以把 Python 代码直接编译成等价的 C/C++ 代码，从而获得性能提升。这类工具有 `Pyrex、Py2C 和 Cython` 等。而从 Pyrex 发展而来的 Cython 是其中的集大成者。

Cython 通过给 Python 代码增加类型声明和直接调用 C 函数，使得从 Python 代码中转换的 C 代码能够有非常高的执行效率。它的优势在于它几乎支持全部 Python 特性，也就是说，基本上所有的 Python 代码都是有效的 Cython 代码，这使得将 Cython 技术引入项目的成本降到最低。除此之外，Cython 支持使用 decorator 语法声明类型，甚至支持专门的类型声明文件，以使原有的 Python 代码能够继续保持独立，这些特性都使得它得到广泛应用，比如 `PyAMF、PyYAML` 等库都使用它编写自己的高效率版本。

安装 `Cython`：`pip install -U cython`

直接拿一个 `arithmetic.py` 尝试一下，执行命令 `cython arithmetic.py`，会生成一个 `arithmetic.c` 文件，而且包含了巨量难以看懂的代码，不过机器生成的代码本来就不是为了给人看的，所以还是交给编译器吧：

`gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -fno-strict-aliasing -I /usr/include/python2.7 -o arithmetic.so arithmetic.c`

等待编译、链接工作完成后，`arithmethic.so` 文件就生成了。这时候可以像 `import` 普通的 Python 模块一样使用它。

每一次都需要编译、等待有点麻烦，所以 Cython 很体贴地提供了无需显式编译的方案：pyximport。只要将原有的 Python 代码后缀名从 `.py` 改为 `.pyx` 即可：

```
>>> import pyximport
>>> pyximport.install()
>>> import arithmetic
```

从 `__file__` 属性可以看出，这个 `.pyx` 文件已经被编译链接为共享库了，pyximport 的确方便。

接下来学习如何通过 Cython 把原有代码的性能进行提升，比如在 GIS 中，经常需要计算地球表面上两点之间的距离：

```
import math
def great_circle(lon1, lat1, lon2, lat2):
    radius = 3956     # miles
    x = math.pi / 180.0
    a = (90.0 - lat1) * (x)
    b = (90.0 - lat2) * (x)
    theta = (lon2 - lon1) * (x)
    c = math.acos(math.cos(a) * math.cos(b)) + (math.sin(a) * math.sin(b) * math.cos(t
heta))
    return radius * c
```

先使用 Cython 的类型声明进行修改：

```
import math
def great_circle(float lon1, float lat1, float lon2, float lat2):
    cdef float radius = 3956.0
    cdef float pi = 3.14159265
    cdef float x = pi / 180.0
    cdef float a, b, theta, c
    a = (90.0 - lat1) * (x)
    b = (90.0 - lat2) * (x)
    theta = (lon2 - lon1) * (x)
    c = math.acos(math.cos(a) * math.cos(b)) + (math.sin(a) * math.sin(b) * math.cos(t
heta))
    return radius * c
```

通过给 `great_circle` 函数的参数、中间变量增加类型声明，Cython 代码业务逻辑代码一行没改。使用 timeit 库可以测定提速将近 2 成。还有一个性能瓶颈，就是调用的 `math` 库是一个 Python 库，性能较差，于是这里可以直接调用 C 函数来解决：

```
cdef extern from "math.h":
    float cosf(float theta)
    float sinf(float theta)
    float acosf(float theta)

def greate_circle(float lon1, float lat1, float lon2, float lat2):
    cdef float radius = 3956.0
    cdef float pi = 3.14159265
    cdef float x = pi / 180.0
    cdef float a, b, theta, c
    a = (90.0 - lat1) * (x)
    b = (90.0 - lat2) * (x)
    theta = (lon2 - lon1) * (x)
    c = acosf((cosf(a) * cosf(b)) + (sinf(a) * sinf(b) * cosf(theta)))
    return radius * c
```

Cython 使用 `cdef extern from` 语法，将 `math.h` 这个 C 语言库头文件里声明的 `cofs` 、 `sinf` 、 `acosf` 等函数导入代码中。因为减少了 Python 函数调用和调用时产生的类型转换开销，使用 `timeit` 测试这个版本的代码性能提升了 5 倍至少。

比起直接使用 C/C++ 编写扩展模块，使用 Cython 的方法方便得多。

> ### 注意
>
> 除了使用 Cython 编写扩展模块提升性能之外，Cython 也可以用来把之前编写的 C/C++ 代码封装成 .so 模块给 Python 调用（类似 boost.python/SWIG 的功能），Cython 社区已经开发了许多自动化工具。